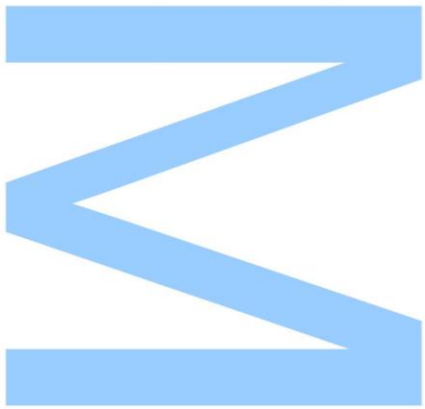
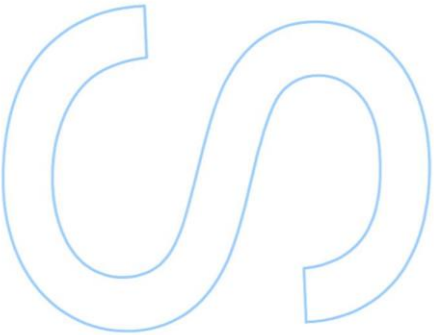
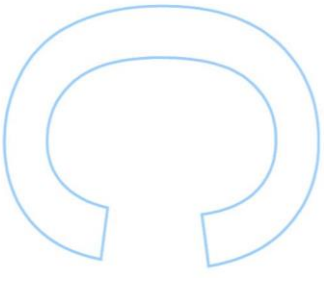
Reconfigurable SmartComponent System







Luis Carlos de Sousa Moreira Neto

Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos

Departamento de Ciência de Computadores

2014

Orientador

Engº Gil Gonçalves, Grupo de Engenharia da Decisão e Controlo ISR - FEUP

Coorientador

Engº João Reis, LSTS - FEUP

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Agradecimentos

A todas as pessoas que influenciaram o meu percurso de vida.

Resumo

A procura pela eficiência como necessidade de rentabilizar recursos traduz-se numa aplicação de conhecimentos em várias áreas-chave da sociedade, entre elas e sendo foco deste trabalho, a Industría.

Ao analisar a gestão de recursos, de uma macro prespectiva, como acontece na Industría, deparamo-nos com um cenário de produção massificada em que o desperdício de matérias-primas e uma ineficaz utilização dos recursos de transformação nas linhas de montagem se traduzem em ineficiência acentuada, nos tempos de preparação para produção (*ramp-up time*) e produção; assim como no desperdicio de matérias-primas e gastos elevados com os meios de produção.

Com base nesta premissa, o objectivo deste trabalho será culminar num componente de software que, em colaboração com vários actores integrantes num sistema de monitorização/atuação sobre linhas de produção industrial, seja capaz de integrar módulos de análise de dados em tempo de execução. O resultado será a manutenção e autoreparação das máquinas transformadoras na linha de produção, a redução do tempo de reconfiguração das linhas de montagem, e a recalibração automática de parâmetros de produção, resultando numa maior eficiência de recursos e tempos de produção.

Por último, resta frisar que este trabalho foi realizado no âmbito de dois projectos Europeus, *I-RAMP3* [1] e *SelSus* [2]. Em colaboração com vários parceiros industriais e cientificos, os grupos de trabalho destes projetos apresentam uma maturação de conhecimento que foi essencial para a realização deste trabalho.

Abstract

Demand for resource efficiency compliant with urging environmental regulations represents nowadays a general concern in many key fields of society. One of those main fields it is industry, where manufacturing processes occur at large scale, requiring strict control to fulfill business requirements, such as demand and cost targets.

Analyzing the resource management, from a macro perspective, as happens in Industry, we face a scenario of mass production in which the waste of raw-materials and efficient management of the manufacturing resources results in a sharpened inefficiency, implying significant costs with production means and production *ramp-up-time*. Based in the previous statement, this work aims to create a software component that, in collaboration with several actors, members of the same system monitoring/controlling the production lines, is capable of integrating data analysis software modules in runtime.

The result, will be the smart maintenance of the production machines, reduction of the ramp-up-time and the automatic calibration of production parameters; culminating in an enhanced efficiency of resources and production times.

For last, this work was realized under the scope of two European projects, *I-RAMP3* [1] and *SelSus* [2] co-founded by the European Comission Seventh Framework Programme. In colaboration with cientific and industrial partners, the working groups of these projects presented a matured knowledge that was essential to this work realization.

Keywords: Industry, Smart Factories, Monitoring, Analysis, Smart Reconfiguration, Services.

"Everyone you will ever meet knows something you don’t."

BILL NYE

Contents

List of Figures

Chapter 1

Introduction

The decreasing price in automation leverages the adoption of advanced machinery as a way to increase efficiency and satisfy constant demands for competitiveness of manufacturing processes. Machines can perform tasks that are hazardous or impossible for humans; their production is consistent in terms of quality and volume resulting in a reduced pipeline. The easiness of automation introduction in factories is due to the fact that tasks at the production line are mechanical and the environment is well defined, the physical area where the production line is contained forms an aggregate of several machines performing multiple tasks at multiple rates in a coordinated way.

In order to achieve the required control over the production process, those aggregates of machines must be managed based on their own working feedback or taking part of the recent growing trend in pervasive and ubiquitous computing systems such as *Wireless Sensor Networks (WSN)*. They make analysis of large quantities of raw data generated during the machines activity and by deploying sensors over the production line or using the machines own sensors, it is possible to collect the data that can be used to improve the production process efficiency. The management of the production process requires a substantial effort once multiple variables can introduce a strong influence on the process, making the production process parameterization inefficient and subject to unexpected influences. These influences can be a partial or total failure from a machine responsible for some part of the production. Unexpected fluctuations on the volume of the production required by the factories customers or a sudden decrease in the quality of the product being made can lead to abruptly schedule another type of

product to be done.

Referring to the previous defined physical area we will employ the widespread term *Shop Floor*, as we will adopt the term *M2M* to represent communication from machine to machine, instead of peer-to-peer communication.

1.1 Problem

In industry field, production lines require permanent monitoring, data generated by monitoring sensing devices is used to verify malfunctions and increase factory production efficiency. Typically, the generated data is not treated in ways to obtain information with the highest level of intelligence; this means that, produced data floods the systems with redundant information. The raw data must be afterwards analyzed to obtain highest levels of information quality. For each task being executed on the production line arises the need of different data reduction methods, these are used for extracting information with an higher degree of intelligence, and data validation methods to ensure information quality.

The process of develop and integrate different algorithms of data reduction and validation, for further analyses, requires changes in the monitoring system at runtime. This process represents a non-trivial problem to solve. Implementing algorithms for these analysis is a task assignable to the factory programmers and personal who are typically aware of the system functionality. Furthermore, systems themselves are not capable of *on-the-fly* integration of pieces of code produced on demand. This means that the system must be modular, allowing for fast deployment of new software modules, with facility in removing, testing and changing the active ones.

In the industrial context, the diversity of devices present in production lines *Shop Floor*, is a relevant concerning problem. Complex machines in the *Shop Floor*, like welding machines, have embedded sensors for monitoring its own work. These embedded capabilities must be employed in the same way that sensing devices sensors are. In addition, the system must assure flexibility that comprises all the heterogeneity of hardware present at the lower levels of the system architecture. This requirement allows for cross check validation between different devices, statistical analyses of the data being produced independent of the specific device being collected and aditional

flexibility taking in account the transient *Shop Floor* composition of resources.

The aforementioned processes occur normally at the **design phase** of a production process schema. Latter integration of data analysis modules, comprising a specific service of a device, typically is not possible to occur *on the fly*. This limitation costs much effectiveness of time to the *Shop Floor* planning, and so, its a gap that we will try to fill, resorting to services virtualization and dynamic software modules wiring. All the present devices in the network, must be seen from a horizontal perspective to the system responsible for managing them; sensors or machines, provides services that needs to be abstracted and represented in a similar way. Targeting specific device services to feed reduction and validation modules typically is hard to achieve. This feature is accomplished by matching between the services from a device interface representation, with the software modules consuming them, that way, the analysis modules can use required device services to process their output. That flexibility must ensure to that, at a **post design** phase all the logic can be reconfigured; if we have a aggregation model being feed with a set of device services, we can manipulate that same set, by adding or removing new elements. Those elements could be models too, so we can have analysis models consuming and providing from each other.

The work orientation, attending to the focus, must follow a Service Oriented Software Architecture and Computing strategy. In the following pages, the focus will regard *WSN* technologies in terms of communication strategies, displacement of devices involved in the architecture and other known limitation considerations. To better understand the generality of the architectures, regarding sensing devices systems, we will further expose concepts of sensing devices discovery and integration, target our objectives and show an overall system architecture schematic.

1.2 Contribution to the problem

Structure of the system, as typically on *WSN* structures, involve three main components, (1) the sensing devices in a low displacement level of the physical architecture, gathering data measures of physical measurable properties from the environment surrounding them; (2) a gateway, that provides interface to the last component, covering a certain area of devices; (3) the server, that treats and collects data from one or various gateways and provides advanced methods of information management and flow. In

this work we will define respectively the typical gateway as *SmartNode* and server as *SmartComponent*. Those two pre-defined entities will represent the generic *WSN's* components, but regarding the functionalities that we will further objectify, they acquire a **smart behaviour** due to their functionalities.

Our contribution will mainly focus on the *SmartComponent*, as that component is present at a higher level, such in physical way, as in logic way. The contribution will attain a logistic of manufacturing production logic. As result, this work should enable different kinds of production processes, to be deployed in concurrency, allowing human or logic supervisors at any given circumstance to change the models and in that way contribute to increase efficiency.

Beyond the scope of the problems that the *SmarComponent* proposes to solve, there's a set of considerations, that from an holistic point of view must be considered for each intervening contributor, for a whole final solution involving all the tiers of hardware and software that have an active role. These considerations, mainly information and control, formed also part of the present work study, as the *SmartComponent* has an active central role in the final architecture it will be firstly introduced in the following chapter.

1.3 Outline

The remaining of this thesis is organized as follows:

Chapter 2 Introduces the context and challenges of *WSN* in industry, the two projects where this work is involved and presents similar works.

Chapter 3 Focus this work within the main components of an Industrial monitoring system, interactions between components, communication strategies and logistic of services.

Chapter 4 Considerations to the design approach of the architecture and technologies used are exposed.

Chapter 5 Details about the architecture components and main functionalities are explained.

Chapter 6 Results are validated against an hypothesis based in a real application scenario.

Chapter 7 Conclusions regarding results over requirements and future work are presented.

Chapter 2

State of the art

Our study of considerations for achieve a workable system, has started first by getting a more suitable understanding of how a reconfigurable manufacturing system is expected to operate, so then we could align that expected operational behavior context with the projects guidelines in which this work is evolved. After we cementing that knowledge based strongly in the work context, we will introduce *IWSN* design and challenges considerations, ending up with the study of works evolving *WSN*, those done in the same and different contexts so we can extract best practices from both and give a complete state of the art.

2.1 Architecture Context

This work fits in the scope of two European projects, *IRAMP3* [1] and *SelSus* [2]; both profit from the use of sensors to monitor the factory *Shop Floor*, the information gathered by those monitoring sensors in the **device tier** is sent *up-link* to the **service tier** where the *Smart Component* align horizontally all devices as abstract services and provides means to use them. There are essentially two kinds of services, **(1) device services** and **(2) complex services**.

First ones (1), represent device's virtual abstractions, in other words they encapsulate devices heterogeneity in a way that makes them homogeneous, they provide *raw data* (the physical properties they are able to measure), in the case of the machines with self-processing capacity they must allow to use machine functionalities, as example,

a welder machine capable of auto-parameterization of welding temperature, must announce that feature and the respective abstraction must allow to use it.

Second ones (2), represent instances residing in the *SmartComponent*. Those instances are created by the **planning tier** agents and are supported by the data analyses modules in terms of validation and aggregation (also prevents database and network floods of information). They should be created in any phase, pre, post or during operation of the system, to form logical groups of services that provide them input and capable of producing to other complex services to consume them.

At the **planning tier**, *machine learning* models will be consuming the aforementioned services that, by their side, are capable of predicting failures from the machines and devices accelerating the *ramp-up time* for the deployment of new products, bringing a whole set of benefits to the manufacturing process. In both projects, the general architecture of the system's is composed of the four main tiers represented in the image below 2.1. **Cloud tier** represents the recent concept of *Sensor Cloud*. Every services and information provided by a factory are exposed at this level, which is an aggregation of multiple *intra enterprise systems*, arising from this set of systems an enterprise factory management feature.

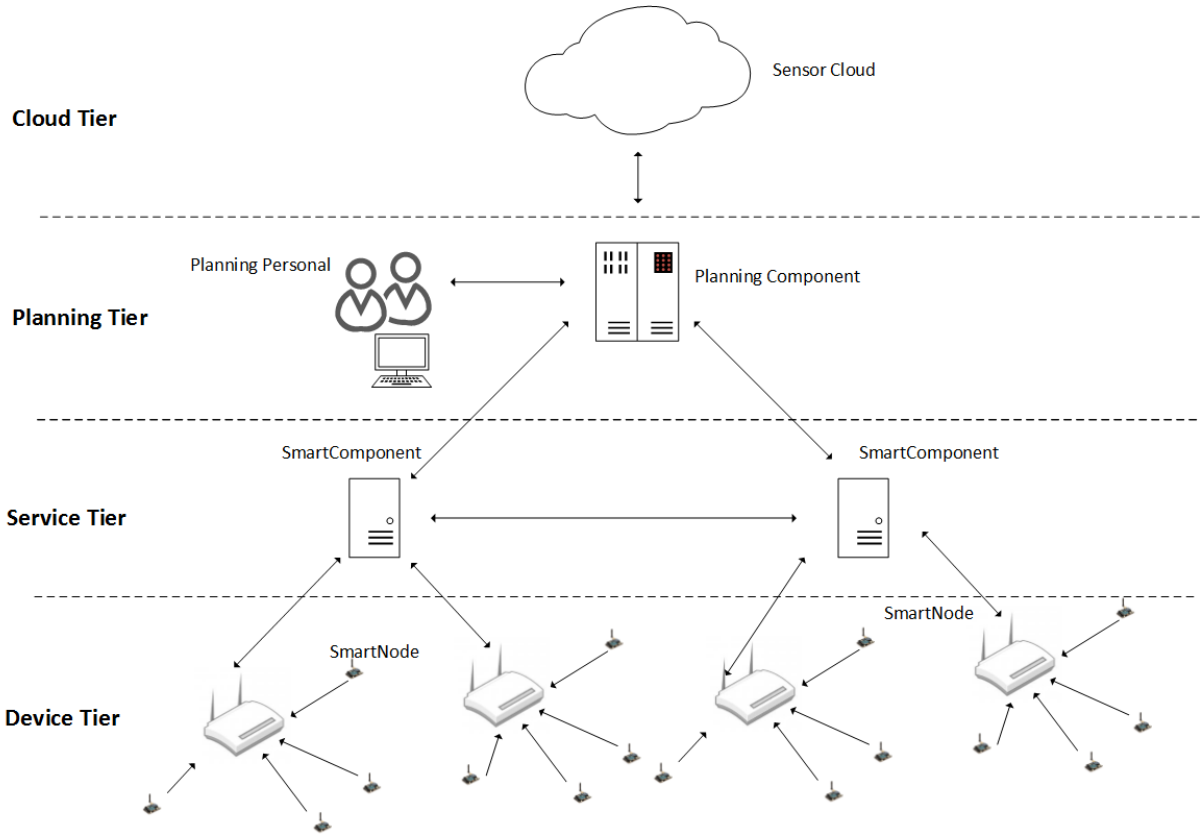


Figure 2.1: Physical Architecture Monitoring Production Line.

2.2 Reconfigurable Manufacturing Systems

Market's globalization brings together product variety and volume needs, turning infeasible old production paradigms. [3] The previous concern arises a new concept of manufacturing systems: the *Reconfigurable Manufacturing Systems (RMS)*. The manufacturing environment, whose two identified influent variables are, product demand and variety; have great impact on those systems, creating new requirements that new production strategies must consider, adapting to environment influence:

- Fast product deployment, for early introduction in the market.
- Product versatility, demands for fast configuration and deployment of system services.
- Market constraints cause the volume of production to fluctuate.
- Decrease product price, implies to reduce costs in production.

Manufacturing activities are composed of three main stages, ordered as Design, Manufacturing and Assembly. Optimizing those stages requires control resources, which must be present to monitor the generated flow of control information. To better achieve efficiency in *Lead-Time* (defined as the time that takes to finish a customer production order), a directly related strategy to the proposed work is the reduction of the system *ramp-up* time. Increasing product variants can be achieved through an effective utilization of the assembly resources, which couples with the aforementioned projects goals. Fluctuation of product demand affects the production platform; this sensibility to the fluctuation can be mitigated preventing failures of the hardware evolved in production process, with constant analyses of efficiency metrics, that couples with the self-awareness and self-healing goals 2.4.

2.3 I-RAMP3

Production system demands 2.2 are the *I-RAMP3* project focus, that stands for *Intelligent Reconfigurable Machines for Smart Plug&Produce Production*. Under the *Seventh Framework Programme* of the *European Commission*, this project involve synergies from both academic and industrial partners; with the aim to give a step forward in the *smart manufacturing systems*.

The efforts of this project working group, focus in the converting manufacturing equipment into smart encapsulated and virtualized devices; referring in the scope of the project as *NETDEVs (NETwork-enabled DEVices)*, turning the manufacturing system components aggregate into a *multi-agent system* capable of inter-device negotiation and production processes optimization. These capabilities will enable important features to the manufacturers, who will have means to an improved diagnosis, *Shop Floor* analysis and smart decisions like scheduled and unscheduled maintenance of the equipment, supported by the own monitoring system feedback. Pointing the main goals of the project, reduction of the production costs, maximum production efficiency - through a fast *ramp-up* phase - and adoption of the *plug&produce* concept.

Sensors have not a significant role in the production process, their impact is mostly reflected in decisions they help to take about the specific machine they are monitoring, typically based in the related machine production life-cycle and safety mechanisms. This project aims to give a significant role to the sensors, they must evolve to reach

a **sensor oriented approach**, enabling process optimization, production automation and more complex decisions over the machinery itself. As example of an advantage in the *ramp-up* process, we have the parameterization of a machine like a metrology system. That process takes a considerable amount of time to do, that kind of systems use light to detect imperfections on parts, like a car hood; slight changes in light can lead to false positives, so the machine must be configured to operate in certain ambient condition boundaries. Moreover, the process is done manually, with a certain number of parameterization iterations to achieve the quality requirements. With the feedback of the sensors coupled to the metrology system and a consequent data analysis, an automatic calibration can be performed, mitigating the manual parameterization effort. Another crucial challenge to overtake is the difficulty in detect a machine wear-out or drift. This project working group associated industrial partners pointed this problem as a major difficulty in Industry; the correlation between virtual sensors groups associated within a machine process, should produce high reliable information, which combined with *machine learning* and *pattern recognition* technics must be capable of detect such unexpected changes in a deterministic way.

Enabling the previously stated functionalities requires - from a implementation perspective - two principal challenges to overcome, (1) well-structured and defined way of communication and understanding between the devices; (2) a smart gateway that allow for integration of different devices, from different vendors and different communication standards of communication (*ZigBee, Bluetooth, IR and other types of RF*). Moreover, every device must have multicast communication ability, so every device in the *Shop Floor* will be aware of the every other devices.

In this project, the communication protocol used is *UPnP* [4], that protocol allows for a device multicast annunciation to every other devices, has notification capabilities and provide means for annunciation of services, respective actions and variables. As this is just a communication mean, to each device extract meaning from the exchanged information, the project defines an ontology based in widespread communication format *XML*, that definition is the *Device Integration Language (DIL)*. Finally, a smart gateway it's used to device integration, the project consortium adopted the *Plug Things* [5] component from the *Freedom Grow* partner. That gateway abstracts the concerns about communication protocols that devices use, it creates virtual representations of

the devices associated to them, which communicate and announce each other via *DIL* and *UPnP*. DIL implements four types that are used in four different purposes between *NETDEV's*, *NETDEV* self-description (*NSD*), describes the device physical and logical characteristic; task description document (*TDD*), used to submit specific tasks to a *NETDEV*, after the analyses of the correspondent *NSD*; quality result document (*QRD*), describes the result of a task after it's execution; task fulfilment document (*TFD*), acknowledge to task submissions and annunciation of the device actual state.

2.4 SelSus

Self Sustaining Manufacturing Systems, is the concept beyond this project whose aim is, once again, explore the concept of smart factories. A special focus on the efficiency concern over resources of production, machines and raw materials will prevail in this project. The machinery is composed of sensible parts that are susceptible to degradation due to high operation speeds, which typically those machines accomplish during the production stages. Due to the previous stated, maintenance in the machinery is a crucial concern to prevent failures and extend its lifetime. Those tasks have regular periods to be executed, based on the time a specific machine is in production and measured since the last maintenance operation performed on that machine, nevertheless, the deterioration of a machine is not linear and unpredicted maintenance operations must be done to avoid a total machine failure. If a machine failures completely, that failure will cause a production process in which that machine is involved to stop. Consequently, mortgaging production time and raw materials that could suffer permanent damage causing waste; in a more extreme case a machine could suffer irreversible damage. All these stated conditions cause a huge negative impact in efficiency of production and economy of the business.

Considering the previous problem as motivation, the vision of *SelSus* is to maximize the machinery performance and lifetime, recurring to a continuous monitoring of production parameters and physical properties associated to the machines being monitored. Monitoring of the machine production process will allow to create patterns of the machine efficiency, thus, the system will make the machine to be self-aware of its own condition. That way if a machine its in a faulty state, timely repair can be done, an unforeseen maintenance can be scheduled with an associated prognosis based on the machine

fault story, as result, accelerating the repair process and avoiding irreversible changes drastically improving the resilience and long term sustainability.

In embedded intelligent systems capable of data capture, the physical link between the collecting devices and the components responsible for receive their data vary. In this project according to the data availability optimal requirements (e.g. *Ethernet, RF, GPRS, optical*). The aggregate of complex network of sensors which performs the monitoring must be **service oriented (SOA)**, allowing for a smooth integration within the **sensor cloud**. This last new concept - that is a requisite in this project as well – will provide real time data sources of information at the factory inter-enterprise level. Expected possible partial breakdowns of the network could occur as the number of deployed devices and the associated degree of intelligence increases; once more, the necessity of the virtualized sensors to form groups representing a specific monitoring service of a machine it's a requisite. In addition, every device in the network should be capable of communicate and understand other devices, requiring well-defined ontology and a communication protocol that allows for discovery, subscription and service invocation. These inter-device synergies will enforce the confidence and intelligence of the decisions being made about the possible errors and irregularities detected at the machines, also eliminating false positives. The intelligence of the decisions will be also product of a collaboration between self-learning modular models of degradation and deterioration analysis, tied together with the components of the network that will abstract each service that represents a machine and its respective association of sensors. Those components from a physical perspective will reside in the *Smart Components*, where this work effort will focus, regarding the project context and taking in account all the previously exposed project characteristics. As a last detail, the technical specifications of this project are not exposed, as they are in decision process discussion from the project consortium. This works aims to give contribute with the study of available and suitable technologies as with the reported results we attain to achieve.

2.5 Challenges of WSN in Industry

This section aim is to expose the challenges that typically are not addressed in the *WSN* general area that regard industrial process automation.

Smart manufacturing systems are the new concept within industrial manufacturing mar-

ket [6], those systems must evolve to adopt intelligent and low cost sustainability paradigms, this opens a gap for introduction of *WSN's* in industry, as this technology has a wide range of applications with known benefits, such low-cost, self-organization, rapid deployment, flexibility and the possibility of embed those micro-controllers in the machinery. Cable cost regarding capital is a preponderant factor (despite the factor that maintenance is arguably considered of lower cost in *IWSN*), such is mobility, scalability and flexibility of possibilities deploying and rearranging the mesh of devices monitoring. As wireless sensor devices are usually small sized, that property enables to measure properties from machines, such as rotating arms and other complex forms of machine operation where the wired devices could not be introduced, resulting in a vastly range of variables that can be measured (eg. image processing, vibration, chock), enriching the information used for analysis and control, consequently an increase in production processes efficiency. A boost in resilience to failures is also introduced with *IWSN*, since the radio communication allows for one or multi hope, dynamic *M2M* communication, we can expect that links from information source to information control will be available with a major probability. However the environmental conditions of a factory, such as corrosive ambience due to chemical processes and high density of machinery and electronic components could represent a harsh environment for a correct function of all *RF* connections.

Discrete manufacturing, products are result of discrete steps, sub-assembly results in parts that are assembled together (Automotive, medical, electronics). That kind of production strategy requires to palletize the sub-products at high speeds to further proceed to the main assembly of parts, the discrete steps are the main responsible reason why real-time needs are so important, to achieve a high speed of production with the desired quality the control over the process must be strictly accurate. Close control loops are usually used in those cases to control the machinery performing the high rate tasks, those are the cases most sensitive to delay, regarding yet interlock mechanisms, that can be used to start and stop the machine activity and to introduce safety control to avoid damage. [7]

Exists a concern with the communication strategies comprising all the layers of the *OSI* abstract stack model *(ISO/IEC 7498-1)*, physical layer that use radio to communicate regarding *WSN* has several adopted standards such as *ZigBee (IEEE 802.15.4)* has

been designed to respect mostly energetic economy issues, that turns to be not suitable for real time automation needs and consequent impact that previous consideration has on synchronization issues. Because of these new standards such *WirelessHART* [8] and *ISA 100.11a* has emerged, with specialized design considerations both in physical and medium access layers. What we can infer about the previous statements, from the experience with industrial partners, is that this issue is not trivial to solve since the devices used to monitor the physical environmental and process properties are chosen based on what is necessary to measure and not regarding the system functional requirements. Based on the previous assessment we could experience some communication issues that can be reflected in the case we have services requiring high rates of data transfers.

In respect to the **availability** the network must be resilient to data errors, such being caused due to a device malfunction or communication issues. In the first case, the statistical analysis of data and establishment of minimum and maximum value thresholds must be adopted to mitigate false positives and false negatives in devices. In the second case, a protocol of communication with reasonable fault tolerance and error detection must be pondered, bad interpretations of data can result in unnecessary system down-times, that represent relevant economy costs in mass production systems. A last consideration, despite the fact that all devices are in the same or in different networks, logically they must be grouped by the production process in what they are involved, this way, a fault in a network ramification does not necessarily affect the state of a concrete production process.

A lack of support to actuators in terms of standards is actually a concerning problem, as aforementioned the gateways must grant the integration of different kinds of devices, this includes the integration of complex machines and actuators. A protocol that addresses this issue should allow a control both payload characteristics in *up-link* for information of monitoring and *down-link* for information of control.

2.6 Studied Works

Several approaches to *IWSN* design were studied in the context of the present work, a distinct approach between our work and all the presented works is the degree of complexity used in the gateway communicating with the devices. Giving more intelligence

to that component, our aim is to concern less about the devices being used and the necessity of flash them with firmware that implements specific functions. That property takes complexity from the devices and adds it to the gateway, while in other approaches a major concern with devices is taken in account.

Firstly, some virtualization strategies are discussed in this work, motes, devices, special services and actors in general are virtualized in upper tiers, then, they are wrapped in bundles. A bundle 4.3.1.1 is a *Java JAR* file that contains a *MANIFEST* file, declaring all the dependencies and capabilities of the compiled piece of bytecode contained inside it. This bundles could require communication with specific actors, they could to, register itselfs into service discovery and provide self interfaces to generic or specific functions. [9]

Exists a trend to create a application stack based on device capacity in terms of hardware. The capacity of a sensor node to process *XML* arise as a weakness that must be solved in a clever way. Author's present [10] a new proprietary protocol to communicate directly between the sensor nodes and the upper level in the stack. While other entities can run existing technologies. Several ways of compression of *XML* are presented, this allow to reduce network traffic and processing overhead. A concern with generic *APIs* for services is exposed, all sensors presented same *API* methods, providing a unique name to distinguish them. For actuators the *API* must be specific, most of all expose different functions with different input parameters to take in account. For sensors should be taken in account the access to change parameters like the threshold of the measures, since we want to remove complexity from the services in our work it should be parametrized in the sensor description. The gateway, referred in this exposed work as "bridge", serves as a translator between upper and bottom devices on the network tiers. The core of this node rely on modules, this modules responsibility is to grant the virtualization of sensors and services to enable translation. In our work we present an additional feature, based in the OSGi model as well, this provides the ability to start and stop software components for eg. data fusion analysis, without the need to reboot the device, enabling dynamic introduction of new functionalities without break connections.

In this approach the architecture comprises a single event bus between the gateway and the *WSN*. This event bus (at the gateway), has three proxy components, responsible for

reconfigurations, a proxy responsible for adding or removing software components, the event proxy add/remove/subscribe, publish and receive. And a third network proxy, responsible for device handling, adding, removing or sending data. It allows to lifecycle management of components and provide means to interconnect masters, that are the components running the core backend of the system. Components are divided in micro and macro components. Some component models for constrained network environments and respective comparison in memory terms to retain are:

- OpenCOM

  Consume significant memory 104KB in the most basic implementation for the features that offers. Use of RPC binding.

- RUNES

  Allows to implement different binding types, implementations just assure RPC. The memory footprint is about 20KB.

- OSGi

  Its possible to manage life cycle of the components, a secure execution environment its granted as run time configuration. The smallest implementation has a footprint of about 80KB.

Communication between the gateways used the implementation of *Jini* [11], event based service oriented architecture, that leverages on *RMI* over *TCP/IP* methods, it has a memory footprint of about *1MB*. This framework was tested in just one type of mote the *SunSPOT*, that has a considerable amount of hardware capacity *180MHz* micro controller, *512KB RAM*, compared to a *TMote Sky 10kB RAM 48kB flash and 8MHz* micro controller. The core of *LooCI* has a footprint of *20.8 KB*, with extra weight added for each macro and micro components of *686 and 587 bytes* respectively. Opting for a cheapest gateway, with more constrained resources this might to cause an excessive overhead.

A concerning problem, it's the use of exactly same services (mean, from the same device) by different models, at a different sampling rate, the one that uses at a higher rate, should provide data to the other one, avoiding to reuse calls and cause network unnecessary traffic. [12]

The possibility of get streams of data is a concern addressed here, the payload size in number of bytes must accomplish all sensors data. Three different sensing device personalities are described, in the basic form of personality, the motes that sense raw data can act in master or slave mode. In slave mode, they act like an expansion of the middleware, communication is done via bus technology. In master mode they are actors of the network, using the radio to transmit the sensed values. The behaviour of the system, regarding low power consumption, is controlled by the motes, if the measurement levels exceed a certain threshold, is triggered a powerful profile. In this profile network capability is increased to support image stream and computing processing of data. Beyond the traditional radio technologies this project uses a wired industrial bus, namely *I2C* bus, ensuring that way faster communication. [13]

This work started to display each network actor providing services, those communicating through binary *XML* format. For real-time needs SOAP is reported as insufficient. Here is made a distinction between internal and external services. Internal, being the services provided by nodes, appearing as services to them respective gateway; external, as services provided by the gateway with more complexity. Its included a component needed to handle all registry tasks, registry of networks, management tasks and node events. This is a backend support to the server, so then, it can locate all external services in all sub-networks handled by the several gateways. [14]

Chapter 3

Work focus

Given the motivation behind the European Industry effort to give a step forward in the smart factories approach, this chapter purpose is to focus the presented work, regarding the projects needs, a match of those needs with the work concerning points will be provided, as a result, we expect to provide a more profound vision over every requirement and solution coupling.

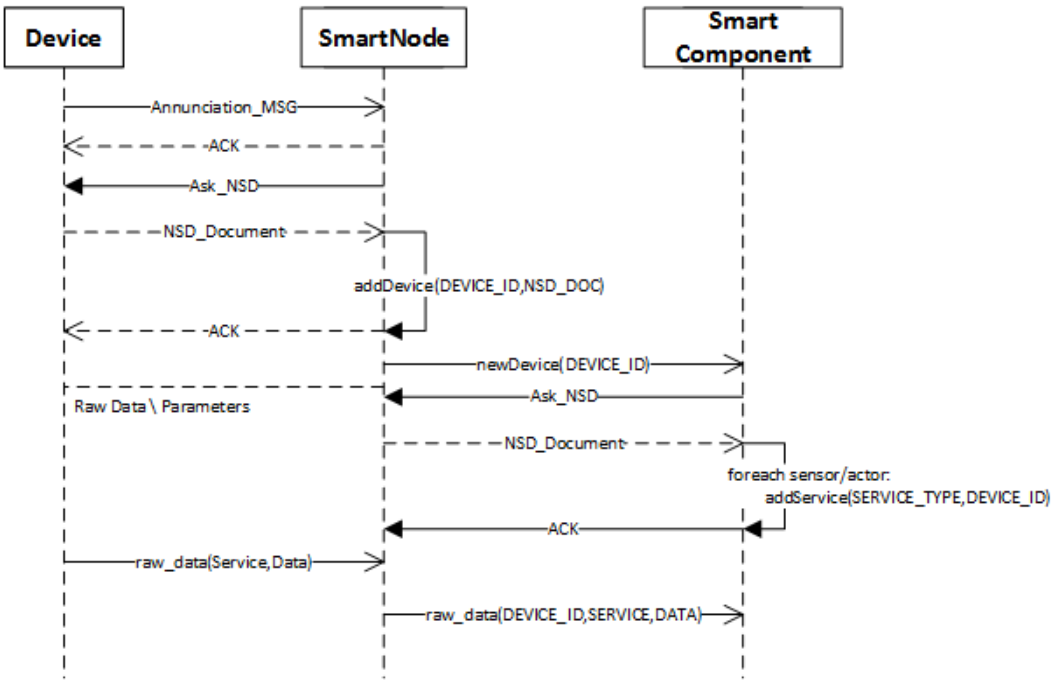


Figure 3.1: Communication General States.

SmartComponent, as seen from a general *WSN* architecture perspective, is the component where the information converges to be processed, covering several gateways (*Smart Nodes*), from that perspective it acts like a *backend* server. Previous figure 3.1, give an insight from where this component will operate, a direct connection with the devices is out of scope of this work, we take the device annunciation to the network for granted, that concern is part of the *SmartNode* component set of functions 3.1. The above presented sequence diagram 3.1 provides a brief illustration of the usual steps that occur since a new device is introduced in the network and detected by the *SmartNode*, until is registered and available to the *SmarComponent*. The interactions between the two above mentioned components will be the priority in regard to the communication concerns of this work, so as the exposure of the inner *SmartComponent* logic management functionalities to the network. This last concern will provide means to other complex components, like factory planning components, to have control over the exposed services and have means to configure and reconfigure the logic of production, through the *SmartComponent* exposed functions. As the system will mostly interact with the *SmartNode*, a detailed description of its operation mode is given next, complementing the *SmartComponent* description.

3.1 SmartNode

Typically the device responsible for data synchronization and data acquisition performs a crucial role in *WSNs*. A smart connotation, is used to describe this component because of the functionalities addressed to it, to articulate the component properly, in the architecture, the next exposed challenges that must be overcome.

- Integration of heterogeneous devices.
- Retransmission of data between devices and the upper tiers.
- Handle communication from upper tiers to device, as well, from device to device.

The logic needed to ensure cooperation between the upper and lower layers of the whole monitoring system resides in this device, in a logic perspective it is divided in three layers, application, middleware and network. The network layer responsibility is to handle protocols, these, capable of assure communication for one side, with devices at the *Shop Floor*, in the other, with the *SmartComponent*. In order to support the

different kinds of physical links, the network layer must handle interfaces for different types of *RF, Optical and Cable* standards. The chosen devices, must vary either due to the needed data availability in regard to transfer rates. The network layer logic must assure synchronization among all the available physical interfaces, it is a complex task due to the distinct nature of the different physical connection characteristics.

A strategy must be reflected and developed to ensure the translation between complex and simple actors involved in the *Shop Floor*. The aforementioned inter-device negotiation property, requires all the present actors to have communication abilities that allow them to interact. Virtually, the complex and simple devices communicate directly from service to service, but at the lower level, who physically ensures the communication is the *SmartNode*. The level of abstraction needed to the upper layers, charge this unit with responsibility to handle the system heterogeneity. Middleware layer, in the *SmartNode* device perspective, is where resides the logic that abstracts from the application layer all the specific network tasks and complexity, providing a generic interface to application the layer and a consequent abstraction of the heterogeneity. Services representation and management, interfaces for parameter establishment and data gathering are requirements of application layer logic. Those previous referred functionalities would be exposed through the protocol chosen to make the connection with the *SmartComponent*, the communication characteristics between that two devices, physical and logical will be based on the *TCP/IP* transport over *802.11* wireless standard, as that devices does not need special physic characteristics to communicate and that way we can choose a reliable standardized protocol concerning just with the services.

The plug of a new device, is a typical scenario illustrated in a basic perspective by the figure above, involving actors from the three tiers of the architecture, thus, also involving communication between these devices, announcement and management of services (respectively in *SmartNode* and *SmartComponent*), and data acquisition requests. Relating this illustration 3.2 with the state diagram presented above 3.1, as the two motes (embedded micro-controller and sensors that forms sensing device) are detected by the *SmartNode*, the smart node asks for them description file, regarding the *I-RAMP3* ontology, a NSD file B. The motes and the associated description are registered within the *SmartNode* and the device is announced to the *SmartComponent*. The middleware

of the *SmartComponent*, during the data acquisition match each data packet received from a device, to the correspondent sensor type. Then the application layer shares with *SmartComponent* the service type keys, sends to that component a packet, that has in the payload the ***DEVICE_ID***, ***SERVICE_TYPE*** and ***DATA*** fields. The responsibility is then delegated to the *SmartComponent*, to match the data with the correspondent service virtualization. On that level, instead of a mote represents a virtual service, a sensor wired to that mote, represents a virtual service associated to a device. This way, the flexibility to use sensor capabilities is enriched and a virtualized sensor can be easily *binded* and used by another services.

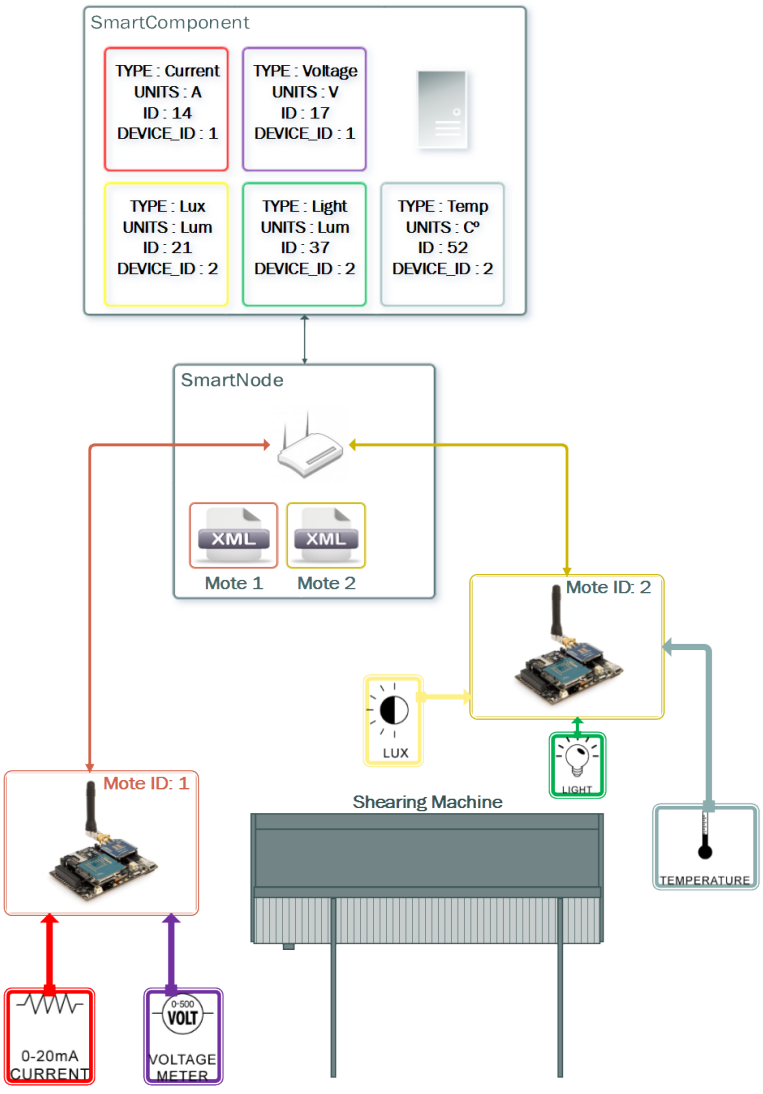


Figure 3.2: Exposure of devices to the network

3.2 SmartComponent

The specific focus of the present work fits entirely in this component, we will summarize the component functionalities regarding the projects requirements, *RMS* previously exposed functionalities 2.2 and other concerns that in a typical *WSN* application are compliant with the upper level component.

In a *WSN* approach, the analogue component to a backend general server is the *SmartComponent*, the work developed in this project is based on that component, regarding the required functionalities that comprises mostly the manipulation of the data collected from the mesh of sensors. Beyond that logic of data management, the objective is to put in this component, responsibility of simple data-analyses functions (what in general approaches happens recurring to devices or gateways processing capabilities), as complex analyses functions. To make the analyses of data at this level, requires algorithms that will perform such analyses, to be present at the time some external entity wants to instantiate such a service, those pieces of software must be easily created and deployed within our application. Considering the data analyses and service management the component must have two registry's:

- Services Registry

  Complex machines, sensing devices and other *SmartComponent* entities will be announced, registered, maintained and discovered through this registry. Firstly the focus will be in the sensing devices sensors, each one representing a sensor abstracted by a service. The following steps will be the utilization of complex machines functionalities (such as control an engine speed) and collaboration between *SmartComponents*. In this last case the services will represent the devices itselves and expose their functionalities through the adoption of a suitable *API*.

- Complex Services Modules Registry

  A complex service represents an instance of a specially purposed built piece of software, beyond data analyses and regarding a *RMS* 2.2 role in the production process management, a complex service can represent control or configuration service. Complex services are built purposed pieces of code or modules, for

later control over individual working cells at the *Shop Floor*. This component must allow to dynamic (on-the-fly), integration of that software components, these, will be kept in the registry of components to match every specific scenario of application. Once instantiated a complex service, the instance will be kept in the same registry, associated to the module that represents the factory for that specific service instance.

The different algorithms of data reduction and validation could be implemented extending the system API, local to the application and distributed or centralized repositories of code must be considered, so the deployment of that modules can be facilitated. Through a web interface of the system, they could be latter instantiated and reconfigured, to consume the different device services or to be consumed by other complex services. The figures 3.3 3.6 explains in a generic way that process. The devices present at the device level, are annunciated trough the *SmartNode*, the *Surface Grinder* and *Shearing* machines are complex device services. In this case they are consuming the associated sensor services data to fed their own service application purpose, the two services representing the machines are, for its turn, associated to the *Door Shape* service. This last complex process was instantiated by an external actor that has previously submitted, the illustrated example of the *Door Shape* requirements to the instantiation. This implies *SmartComponent* to expose methods to allow for the service management and to implement a structured ontology that allows to extract the meaning of the process requirements.

The concept of *Sensor Cloud* emerge at this level, all the components of the *SmartComponent* system, whether they are physical devices or logic components of data processing, may appear as similar and horizontal services. These must be exported to the network in a way that allows represent them into the *Sensor Cloud*. That cloud application, should be able to subscribe the exposed complex services, when adopting a protocol to the communication between the architecture tiers, a publish/subscribe mechanism must be considered.

A generalized protocol, as seen in studied works is the *UPnP*; usage of this protocol provides means for transparent communication between heterogeneous systems, being those systems, actors present at the *Shop Floor*. Each device can implement a template of a *UPnP* description file, that allow for describing the functionalities they met. Regard-

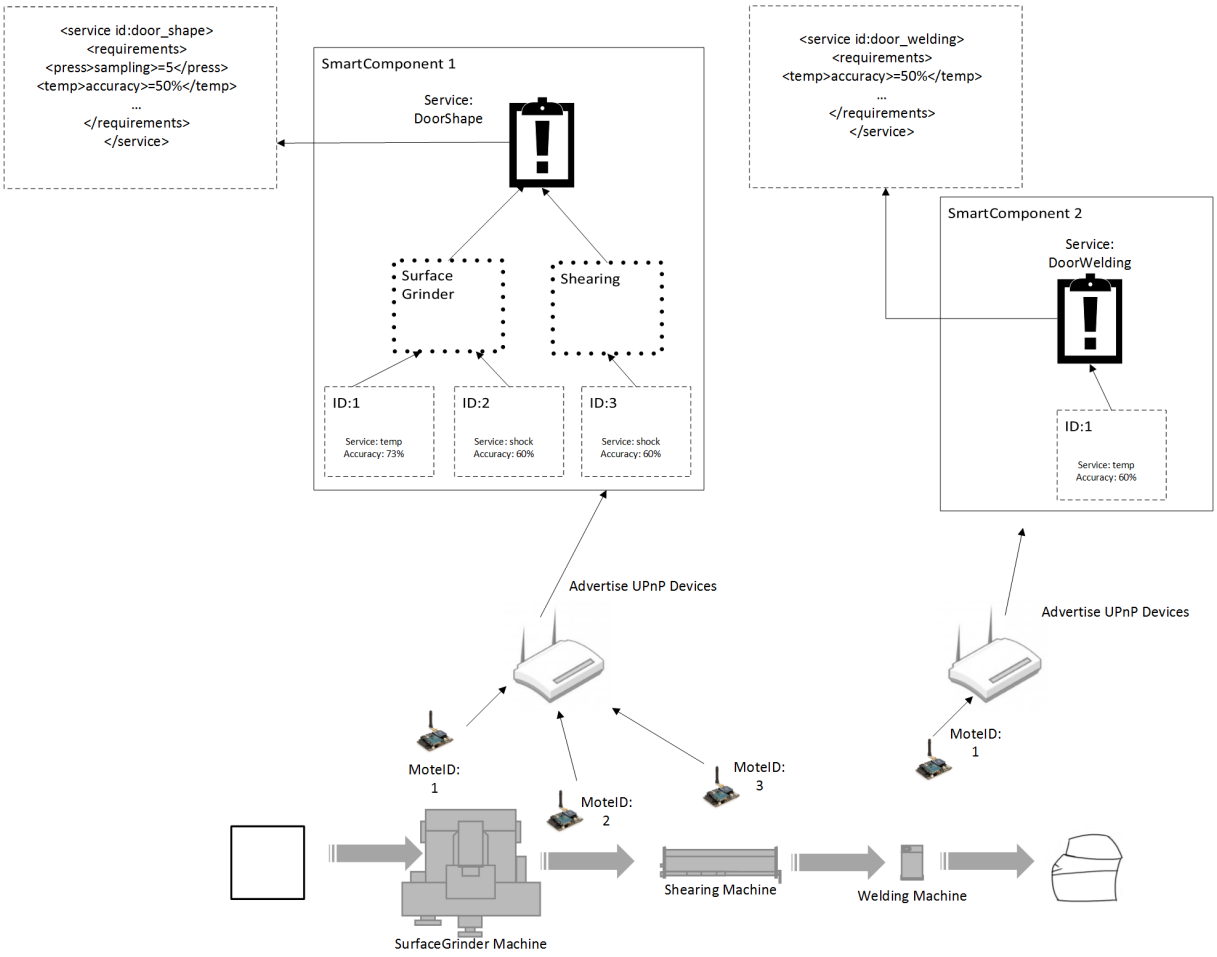


Figure 3.3: Description of Production Services

ing this purpose, each service provided by our system could be represented through *UPnP* device, visible to all the *Shop Floor* network. To a most detailed description of the device and its services, *DIL* language must be considered, and a generic method present in all the network exposed services to get de *NSD* description of that service B. The *UPnP* template file together with the *DIL* implementation, provides transparency for those services being consumed by other systems, even machines, becoming this major importance feature in the system solved.

3.3 Devices Communication and Integration

Communication between Sensing Devices and Smart Nodes happened in a low physical level of the overall system, a scenario in which several Sensing Devices would be deployed; we must expect problems in access coordination, volume of data transac-

tions, few computation capacities from Sensing Nodes to process messages incoming and outcoming.

3.3.1 Communication inIvolving complex machines and SmartComponent

Machines at the *Shop Floor* contains its own communication means, physical interfaces, communication protocols and application protocols. Requiring from the gateway between them and the *SmartComponent* a flexible network layer, allowing for integration of new communication modules, that ahead, will allow a translation to a protocol that turns transparent the communication between all devices in the network. A study of the suitable protocols that allows for an abstraction of devices and at the same time a representation of their services, with the possibility of manage these in a cohesive way, will be of the major interest in this work. Reached that goal, costs of time and money, required for reprogramming the gateway network layer, can be avoided by a simple integration of communication modules trough the client interface at management level of the factory. Consequently, a translation to a single protocol that regards some *SmartComponent* considerations will traduce in *plug&produce* facilities.

3.3.2 Communication strategies

Communication language and strategies, are the two most concerning problems regarding to achieve reliability in the industrial context between *M2M*. Communication strategies, because of Sensing Devices heterogeneity, must implement different solutions in some of the *OSI* model layers. In the network layer, is not usual to employ traditional Internet IP protocol, because of devices low capabilities, this layer is most of the times *6LowPAN* or *NoIP* using proprietary communication protocols.

3.3.2.1 CoAP

CoAP protocol assure a low effort in achieving communication, the binary translation from *HTTP* functions turns this approach a lightweight request interpretation form by Sensing Devices. The inclusion of *CoAP* library in sensors modules, allows for ease en-

capsulation of data and request handling in the two devices. Still requiring modifications in Sensing Devices. Data can also be encapsulated in Sensing Devices proprietary messages protocol, this way, the decoding messages process, will become more cost effective. However, will require specific interfaces design to each type of mote, turning this approach unfeasible.

Related Works

Discussion of results obtained using *CoAP* protocol to enable *M2M* communication for supervision monitoring of environmental conditions. [15]

Using implementation of *libcoap* (*C* implementation library of *CoAP* protocol), to enable *CoAP* over *UDP* in *Contiki* and *TinyOS* embedded operating systems. This specific application requires communication between the gateway and the server trough cellular communication. *CoAP* client and server implementation deployments must run in WSN nodes and the gateway. To achieve communication between backend server and the gateway, three strategies can be employed:

- Proprietary M2M protocol of cellular networks.
- IP application protocols, gateway can act as a proxy. CoAP frame header enables description of the message type, the optional headers are:
- Confirmable (CON) messages always carry a request or response and require an Acknowledgment (ACK).
- Non-Confirmable (NON) messages are used for streaming communication and sampling messages that do not require an ACK (e.g. subscriptions of reading a sensor at a required rate of sampling).
- Acknowledgment (ACK) messages acknowledge CON messages and must carry a response or a null payload.
- Reset (RST) messages are sent in case a CON message is not received properly or some context is missing.

Porting libcoap to *Contiki* and *TinyOS* operating systems process is demonstrated, low effort required for adaptation (*TinyOS* contains the implementation of *CoAP* protocol, coap blip), some optimizations were implemented in order to fulfill memory constrains

requirements of stack and flash memory. Mapping resources requires a pointer from *CoAP* to the correspondent resource interface. The packet size limit that *IEEE 802.15.4* standard defines is only *127 bytes*. IP based application protocols that enable communication between heterogeneous machines, such as *HTTP*, *SOAP* and *REST* demand optimizations to shrink messages so they can serve the purpose.

Metrics reported used in the evaluation tests:

- Response time : Time taken from sending the *HTTP GET* respectively the *CoAP* Request from the client until the connection is closed.
- Total number of bytes transmitted : Total number of bytes transmitted within the above mentioned response time.
- Overhead of the Header : This shows a separation of bytes in each layer.

The unique access method close to *CoAP* results in the two first measures was *HTTP* over *UDP*.

- Response Time:

  CoAP: 1.029 (sec)

  HTTP over UDP: 1.104 (sec)

- Total number of bytes transmitted:

  CoAP: 107 (Bytes)

  HTTP over UDP: 132 (Bytes)

Standard of *IETF* to *CoAP* and *REST* architecture deployment in constrained networks called *CoRE (Constrained RESTful Environments working group)* [16].

3.3.2.2 Zeroconf

Aiming home automation, objectives of this work are to obtain a seamless wrapper (driver), for all the sensor devices, discovery of services provided by sensors and implementation of *UDP/TCP* sockets to interact with the application level. Support to interaction of services, offering synergies across different technologies. [17]

The middleware produced is based in four main model components:

- Service Factory Listens the network for new devices and services and creates the virtual instances as representations of the services available at that domain.
- Virtual sensor Instance that virtualizes the physical sensing device, uses proper native communication protocol, to forward messages in the two directions, achieving the required flexibility for heterogeneous sensing devices integration.
- Virtual Service Represents each service provided by each device, this module Auto-registers the service in the *DNSSD* and listens for requests to this service creating a protocol adapter to application level.
- Protocol Adapter Provides generic interface to seamless interact with the different devices, it can provide several standardized protocols.

This protocol provides automation of three core network services:

- Name resolution
- IP addressing
- Service discovery

In a traditional *Zeroconf* deployment, each network device maintains its own list of available services through the network, the *DNSSD*. Here the gateway is responsible for handle this list; keep track of devices and services they provide.

The flexibility of this work may provide some concepts important to our work. The capability of use multiple protocols to communicate with application are important to relay on the service personality. *Zeroconf* seems to be less complex than *UPnP* approaches, providing the same important core services. The sensor wrapper, that in this work needs intervention from the programmers could be eliminated, employs *OSGi* technology to provide easy deployment of native sensor protocols.

The evaluation was made contemplating just one type of device, the *SunSPOT* platform microcontroller and no metrics or other concrete results were showed. It's referred that the middleware is lightweight but no arguments were presented to support that conclusion. *Zeroconf* protocol is under standardization process, *IETF* and *RFC* formed working groups working on these standards, which makes of this protocol a solution to

consider in our implementation.

3.3.2.3 RestFull

In a strictly *REST* architectural style implementation, services must be stateless to handle the several parallel calls at the wide network. Applying this philosophy to *WSN*, the lightweight approach compared with *SOAP* style is a valuable advantage. While in *SOAP* messages must be transmitted in a *SOAP* envelope, in *REST* messages are *XML* or *JSON* conversions of data. We can take advantage of this factor to allow for periodic sampling subscription and streaming of data without overcharge the network channel. Resources (abstractions of services), are represented by *URI's*. *REST* uses the *HTML* verbs *GET, POST, PUT and DELETE* to manipulate the resources, those four operations are sufficient to manipulate the required common actions in *WSN's*. Following illustration conceives the general idea behind.

The following table illustrates the utilization of *REST* calls to interact with services provided by different components. I this case, invocations to the *SmartNode*, allowing to obtain information about the devices underlying the component.

URI	GET	POST	PUT	DEL	Result	Device
\	X				Service Collection	Smart Node
\SERVICE_TYPE\	X				SERVICE_TYPE Collection	Sensor Device
\SERVICE_TYPE\ALERT			X		Subscription(Threshold)	Smart Node
\SERVICE_TYPE\ID\RATE		(X)	(X)		Subscription(Rate)	Both

Figure 3.4: RESTfull calls.

Abstract the numerous proprietary protocols needed to be handled by programmers. Provide service management and coupling in a dynamic way. A middleware hiding the multiplicity of service discovery and communication protocols is built according to this model. Tools and techniques are necessary to hide the heterogeneity of the service protocol. This work relies on home automation project, enabling media streaming over the house, following house owners and matching streaming to them behavior.

The solution is built on top of *OSGi*, *Java Reflection* is employed to match proprietary

protocols with a set of mapped proprietary protocols and generate the appropriate driver. It is said that, bytecode generation of the drivers is a better approach generate code at run time. [18]

UPnP is used for service discovery, and registry. Closed to this a ontology for describing services requirements and description is employed. This strict ontology allows for services requirements to being matched trough a sorting algorithm that binds the appropriate service to the required conditions. Solving the protocol heterogeneity we would face an interface fragmentation. The idiosyncrasies between devices cause a fragmentation that sensor driver at the top layer must handle.

As seen in previous works, service adaptation consists in providing means to semantically similar services to be accessed by service consumers through an adaptive generic interface. This interface can be a proxy or a virtual adapter. This can be done in two main steps, service matching following a service adapter generation. Here its employed *Java reflection* to generate the appropriate adapter, and the match is done by means of an ontological mapping of definitions.

Building smart automation trough *IEEE 802.15.4* devices trough *6LowPAN* addressing and *RESTful* based service management. [19]

URI's represent the resources access. The available services are categorized as follows:

- / - The root collection of sensors
- /temperature/ - Collection of all sensors providing temperature in the domain*

JSON is used to represent serialized data because of the lightweight compared to *XML*. For reliability of data, *HTTP* is stacked over *TCP*, is argued that persistent *TCP* connections can achieve a significant reduce of overhead. A *JSON* formatting library is employed to overcome manual typing errors and performance enhancement. Whenever a data stream subscription is required *UDP* is used to transport data. The memory footprint of the compiled application deployment needs a flash capacity of *43 Kb*. In those, only *2 Kb* are correspondent to the *RESTful API*. The difference time response of a *TCP* established and closed connection is significant, the first registering an average response time of about *180ms* and the second over *260ms* for a *94 Bytes* packet.

Applying *REST* to constrained devices architectures, is reasonable to maintain several rendered components and service information, this approach can save a great computation effort in a rebinding caused by a change required by a ranking match of services.

This approach is very reliable in terms of performance such memory amount needed and response times. The *6LowPan* stack is a inflexible approach to our work, but with an appropriate flexible stack, the rest of the technology would be a good approach. Technologies *mDNS* or *Bounjour*, could be useful to send multicast service requests between Smart Nodes. *Java reflection*, generation of code after matching a service method, by name, arguments an output. Code can also be generated through bytecode, better approach for runtime deployment.

3.3.2.4 SOAP

In architectures purely *SOAP* based, sensors register themselves within the gateway, gateway requests the *WSDL* file and maps it to sensor device *ID* (eg: sensor IP), representing the *UDDI* entity. Clients then query gateway for *UDDI* available services, and trough *SOAP* messages can connect efficiently the individual nodes services. *SOAP* message envelop transact data in *XML* format; typically the proprietary frames structure payload is small so it's necessary to fragment the *SOAP* message requiring the transmission of more packets. Without a more profound investigation we can easily affirm that for real time needs of communication *SOAP* is an inefficient solution. We could implement more logic to outline this problem, but will result in more complexity to the gateway. A descriptive diagram of the *SOAP* architectures applied to WSN overall solutions is illustrated next.

Access from IP based networks to *WSN* enabled by Device Profile for Web Services (DPWS) was explored in Home and Industrial context. The advantages of using *6LowPAN* are detailed in this work, the possibility of have an link local address for unicast and yet an *IPv6* interface, coded associating the coordinator ID to the link local allowing for direct communication with other devices under different coordinators. [20]

6LowPAN provides 4 frames, those could be very useful for protocol implementations since defined standardized frames are divided into:

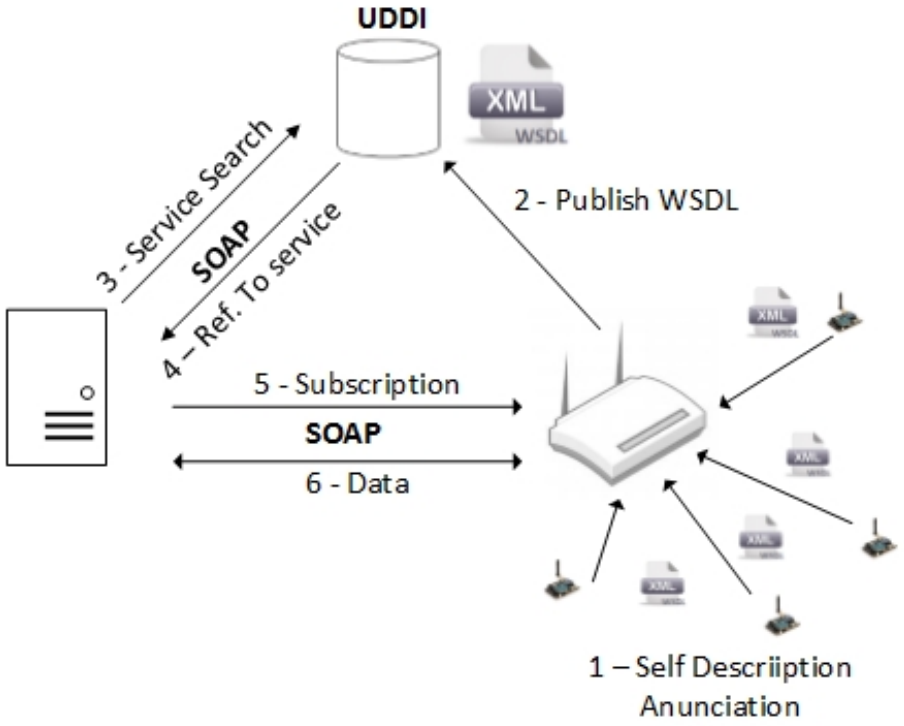


Figure 3.5: SOAP WSN architecture.

- Data frame (raw data)
- Beacon frame (annunciation)
- MAC command frame
- Acknowledgement

Heterogeneity of devices prevents the use of *6LowPAN*, the *Shop Floor* industrial line machines used for production are a heavy factor for the need of a solution that embraces proprietary protocols. No tests in this presented work were made, industrial context can be eventually a challenge in terms of performance requirements.

3.3.2.5 RPC

Remote Procedure Call based solutions can achieve some key requirements this work aims to fulfill. In *RPC* are included *RMI* and *SOAP* solutions, but we opt to specify them apart because of the numerous works deployed using those two approaches. In *RPC*, client just needs to know the interface description of server methods; the commu-

nication agreement is conceived based on methods description files and a stub in both sides. Depending on the problem approach, different data marshaling technologies could be employed, over different transport and application protocols.

3.4 Management and reconfiguration of services

In a logic perspective, two types of devices represent services, machines and sensing devices, they must met at the same level of abstraction, since we could couple simple services (those provided by basic sensing devices) and complex services (provided by complex machines). This coupling results in integration of new data processing models, feeding them with specific input requirements, results in valuable information, that will be of the highest importance for the *Shop Floor* planning. The richest the information provided by the data analysis modules, the most efficient the plans to industrial lines, these becoming more cost effective. The services of the devices connected through the gateway, need to appear to our component in a horizontal perspective, allowing for a flexible management. Management can be achieved recurring to a ontology, that is applicable to all the architecture actors, so they can extract valuable knowledge from each others services output. The implementation of a richest ontology model that regard the previous purpose, represents a contribution to the final solution that is of our best interest. The image below ilutrates the horizontal perspective of complex and sensor services. The acronym *VSIG* stands for *virtual sensor information group*, it denotes an aggregate of virtual services, that can be managed to feed a complex consumer service. The arrow in the image denotes an rearrangement of the services logic, the service **1** was eliminated, service **2** was reconfigurated and the service *3* was instantiated and the respective *VSIG* of providers associated.

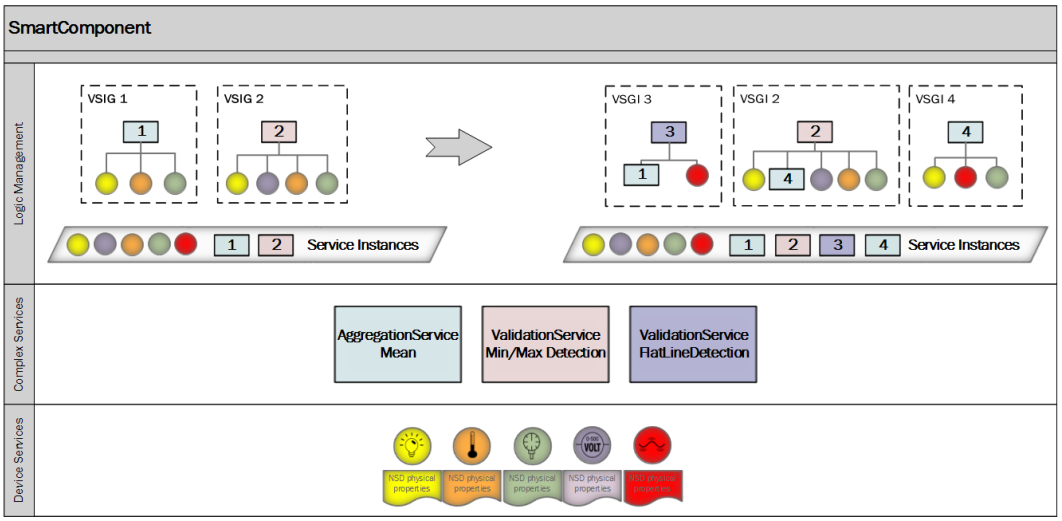


Figure 3.6: Reconfiguration of Production Services

3.5 Contribution to other works

For grant specific monitoring requirements over the machines, properties like accuracy and percentage of area of that machine that is covered by a sensor are relevant configuration indicators when post designing a production process. Regarding this necessity, a parallel work, developed in the *I-RAMP3* project, was integrated within the *SmartComponent*. The integration was a helpful process that highlighted design needs of the architecture to future acomodation of other components. Coverage, connectivity and requirements of a *IWSN* are the directions of focus of that work. It functionalities make it possible to act in the configuration and control phases of an *RMS*, specifically, revealing to be a relevant help in the planning phase of a *IWSN*. Interacting with the services present in the registry of the *SmartComponent*, its possible to calculate if a the number of devices present of a specific type (eg. Temperature), with a given coverage radius is enough to cover the area where the machine to monitor is present. This software component abstracts an algorithm that is in its core, this algorithm is capable of solving the proposed problems of area coverage, allowing to introduce and treat obstacles in the area. This way, in real time our work provides information to process the results that latter will serve the purpose of network configuration at the *Shop Floor* [21].

Chapter 4

Methodology

Considering the premises that constitute the work objective, the effort applied in attain a solution will be divided in two main effort topics, interaction between network actors (low level) and logic services management (high level). A first approach step taken, was the study of existing works in the sensor integration topic (as presented in the second chapter). Perceiving the communication strategies adopted to solve heterogeneity in device integration, was a significant contribution to perceive communication between all the entities in a transparent way, as main limitations and concerns. As second step, we adopt the Service Oriented Architecture methodology strategy, to target the necessity of a flexible logistic of services, in a way that we can create, reconfigure and deploy services with a high degree of flexibility. The third and last step, was the study of technologies suitable to the problem, regarding the previous two steps acquired considerations and mostly, the adopted technologies and methodologies by the projects working groups.

4.1 Service Oriented Architecture

A software application that implements the Service Oriented Architecture (SOA) paradigm is typically designed for web or corporations, those represents large and distributed applications composed of services that all together constitute the set of the application functionalities. Services represent single pieces of code that abstract its core logic, providing a specific set of functionalities and implementing an interface that exposes to other services the meta-information they need to invoke and understand each other.

The framework of a *SOA* application is formed by the components that orchestrates the services, that core of components ensures the services interact to meet the application goals. Services must be loosely-coupled, services can be easily replaced, coming and going in an unpredicted way, they just need to know how to operate under its own interface, having no dependencies with external services. As *SOA* was not originally created for application in our work context, we will expose next some related approaches in the same context, that addresses the *SOA* concept to *IWSN* applications. The goal of the following SOA guidelines for a *IWSN* is to achieve a cooperative, adaptive, scalable and data mining capable, network distributed application.

The two concepts, (*IWSN* and *SOA*), have compelling incoherencies, a *IWSN* must have a deterministic behavior; the service discovery component, a fundamental component of a *SOA* application, is generally not deterministic. Regarding determinism, the application must issue warnings in the absence of any required service, if a specific service does not exist, the application must delegate a service of the same type. A service of the same type, regarding sensors, means that if a consuming service searches for, eg. a temperature service, with a specific accuracy or from a specific device. If that service does not exist, the application should couple other existing temperature service with similar properties to the one that will consume. Part of the orchestration components of a *SOA* is **service registry**. In the studied approaches the registry must be aware of the available services provided by devices, in our case, beyond that task, registry must be aware of the instantiated complex services (eg. validation and aggregation).

One challenging aspect addressing *WSN* to a *SOA* is the absence of reusable technologies, tools, components and proper standards. In every works studied to compile a *SOA* approach to this work, different technologies where employed, the challenges faced in every work have slight differences due to the components and the authors raise this necessity of a necessity of standards. [12]. Designing a monitoring system using a *SOA* architecture approach, oriented to the industry, was previously done by the project *SOCRADES* [22], one of the resulting publications of that project consortium has highlighted the main challenges in design such an architecture, regarding good practices of software engineering, we will expose next those practices. The first two considerations are based in the sensor nodes, both hardware and software, in this

concrete case, design software to the devices must consider to have low footprint, low overhead and cares about power consumption. As the *SOCRADES* project use the same type of devices with different sensors coupled, this approach implies the industrial companies to buy specific hardware and does not allow for integration of different types or devices and complex machines. Regarding differences in hardware (vendors and models) choice, due to different industrial partners involvement and integration of the existing hardware in the Shop Floor our aim is to grant flexibility so we will not address that concern.

One of the fundamental key aspects in *SOA* applied to *WSAN* is the need of capability to easily handle the system heterogeneity. It allow a smart, flexible management in adding or removing devices, actors and make services interoperable. Exposed as a requirement, this is a crucial feature to enable the device interoperation, communication among all the tiers of the architecture, this way creating synergies between all the actors enforcing the confidence on data and better diagnosis, prognosis and self-awareness. Considering the device level of the system, is raised the importance of suitable device encapsulation, that property is essential to grant a collaboration of all the devices. As we are virtualizing devices to announce them to the upper layers of the system, a virtualization corresponds to a service, which is in turn, the encapsulation of a device (ie. a *NETDEV*), is this abstraction in the *I-RAMP3* project2.3. The way a service exposes its own properties and functions - that must be generic to all the services - makes the abstraction more or less suitable regarding the application needs. The service properties must be filled with the corresponding device specific properties, a device has analogue and digital inputs, this connectors allow to wire different kind of sensors measuring different properties. With a considerable focus on sensors (types of sensors, units of measure and accuracy), each sensor must be considered a different service, so when it comes to the device describe himself, it must further than publish its own characteristics, announce the coupled sensors, respective units of measure and accuracy of measurement. To fulfil this need, *DIL* language will have a significant role, as the *NSD* description document describes all the associated sensors and its properties B. The *NSD* document can be flashed within the device, using a specific method to be invoked and return the *NSD*. In the case the device has not enough processing capacity to do that task, we assume the *SmartNode* will interpret those

capabilities and produce a respective *NSD* document.

Applications dealing with constrained devices, scalable and dense networks must be based on asynchronous rather than synchronous processing. An request to a device must be a case of special necessity (eg. request a snapshot measure of a sensor), if the application is continuously polling data from sensors rather than triggering event handlers when the sensor sends data, it will flood the network with packets flowing up and down link.

As previous inferred in the projects context, validation, aggregation, control and configuration software components must be accommodated by the *SmartComponent* architecture. This work focus is the architecture, however, we will focus also in the integration of validation and aggregation modules, this way we will better prepare the architecture for later integration of other software components, such as configuration and control ones. A considerable difficulty in make a system sufficiently generic to accommodate different application modules was reported also during the study of similar works. Relying in the studied works, dealing with this dynamic of application modules was succeeded using the *OSGi* technology, capable of runtime integration of software components it reveals as the chosen technology to develop this work.

4.2 Service Oriented Computing

The SOC concept presents a methodology that aims modular software components, heterogeneous and autonomous between them, resulting in a software architecture oriented to service. A service provides a higher degree of abstraction, thus turning to be the best option for large scale applications.

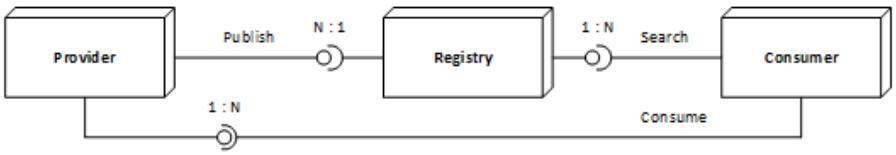


Figure 4.1: SOC three main components.

Rather than focus on the standardized approach of web services, our aim is to develop an modular architecture for a service based application, where the three main components will prevail with an emphasis in computer science approach following the

elements of the correspondent engineering methodology. *SOC* considerations focus on four levels of abstraction depending on the architecture cross levels, those concerns are based on the services within the application (inside *SmartComponent*) and aspects regarding interactions across enterprises (between actors, *NetDev* abstractions). For each one we will retain below those of most interest.

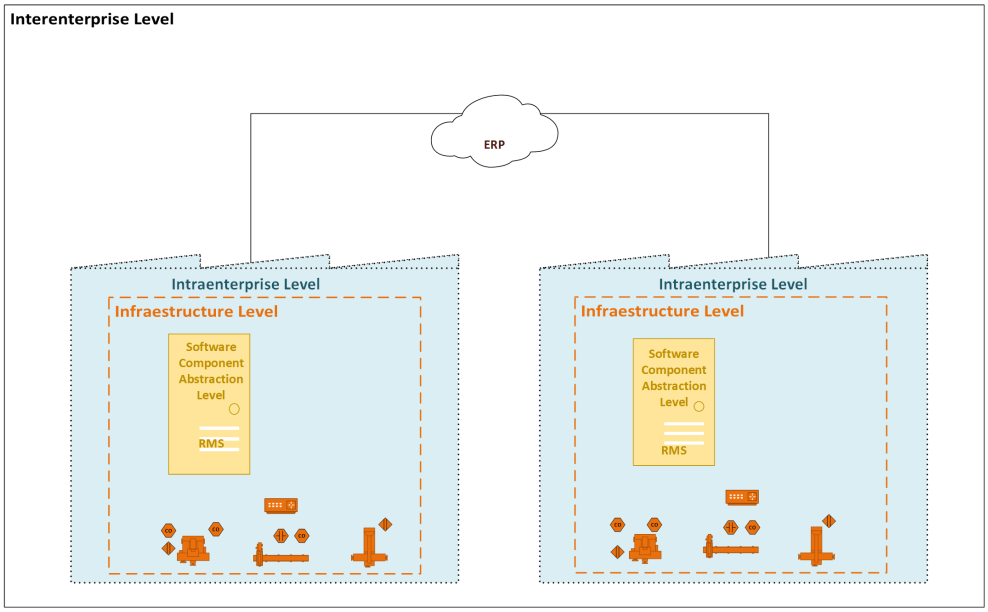


Figure 4.2: SOC levels in industry context.

- **Intraenterprise Level**

  Interoperation between actors at the *Shop Floor* represent this level of abstraction, concerns at this level are the protocols being used to ensure communication and utilization of defined accurate declarative information models so we can reconcile interacting components. To meet a requirement such as integration of a new application, the utilization of intercommunication standards is imperative, rather than that, it must expose interfaces that relate new application data models and guarantee transactional properties regarding the intra-enterprise properties of the organization they represent. In an operational and industrial context we can assign this level to the *RMS* concept, as the agents in this scenario are in the factory *Shop Floor*, our aim is to make them collaborate in order to reconfigure their behavior, so they must communicate and understand each other in order to exchange services. Requirement for protocols to communicate through the different component abstractions, in our case *UPnP* will provide the means to communicate and create the synergies between the Shop Floor actors, each actor encapsulated by a similar *NetDev* service. A data model format to extract meaning from the communication will be achieved by using the *DIL* schemas, a defined ontology adopted in the *I-RAMP3* project, based in *XML*. The encapsulation in *NetDev* abstractions and the use of a predefined language will allow for the focus of *SOC* in this level, communication in a uniform and transparent way and easy exposure and uncover of application modules inside the factory.

- **Interenterprise Level**

  Supposing that more than a party must be involved in the design of the application, this level of abstraction cares that inter-enterprise level of considerations, those at this level are a certain degree of fault tolerance and a policy for rescheduling transactions and rewiring inter-enterprise software components. In this work case those considerations are beyond of the scope of the objectives to be met, but regarding the *SelSus* project, a *Sensor Cloud* would be present that aims to exchange service functionalities and data across large organizations, *UPnP* and *DIL* schemas will enable a easy integration of the *SmartComponent* services in the cloud, consequently supporting the *Sensor Cloud* purpose and further enabling to use *ERP (Enterprise Resource Planning)* that are systems whose

aim is to control core business process through large organizations; they are used to exchange data among departments so production planning, inventory and economic activities.

- **Infraestructure Level**

  A strong analogy to the infrastructure level are the Grid Services, these services must be unware of the infrastructure that is supporting them, infrastructure components come and go in a unpredictable way and that behavior raise an interest in modular interfaces based on services. Another key concept emphasized at this level is that of ***utility computing***, a *SOC* architecture must be configured dynamically and *on-the-fly*. Service instances must embrace that dynamical behavior, allowing to create and bound instances as needed. Resources identified in the image above as part of the infrastructure, are all of those evolved in the production and monitoring process, machines, sensing devices and gateways. Resources can abruptly leave the network or be idle as they can be included, regarding a aforementioned *plug&produce* way. The architecture must expect and treat sudden changes in the infrastructure, in a way that regards the loosely coupled connection to the hardware resources and other services depend on the resources such is the case of sensor services instances.

- **Software Component Abstraction Level**

  Creating new software components for the considered architecture is a tak that fits this level of consideration, a clear interface semantic for services as for the components should be exposed. The task of developing new software components using that interfaces result in an easily customization of the software. New types of machines and sensors as new software requirements for data analysis will arise, relying on the previous statement, development of new software modules to embrace new hardware characteristics and data analysis techniques should be a concerning feature. As we could observe in the image, at this level we will focus on *RMS* systems concept, the system must provide new means to enrich the logistic of the production management logic, what can result in new software modules or modification of the existing ones.

4.3 Physical architecture

4.3.1 Technologies

Regarding the aforementioned objectives of the architecture, the heterogeneity of devices and previous studied works, *Java* is the chosen language. *Java* is a *OO* cross platform programming language, the compilation of Java code results in bytecode, that runs on the *JVM (Java Virtual Machine)*,the component that turns *Java* to be hardware independent. This language targets all the requirements since the adopted communication protocol (UPnP) and system service modular capabilities (provided by *OSGi* specification) are achieved by the respective specification groups.

4.3.1.1 OSGi

Stands for Open Service Gateway Initiative [23], the original focus of this specification was the deployment of dynamic modular service gateways, the wide range of applications that a modular system can provide, later, makes this specification to evolve and integrate projects as the *IDE's Eclipse* and *NetBeans* or the *Red Hat Application Server JBoss*.

Dynamic integration and management of components occurs through a lifecycle, the *OSGi* components, bundles, run on the used *OSGi* framework, the framework acts as a broker between the *JVM* and the bundles. This framework manages the bundles lifecycle and dependencies, this means that a required bundle to deploy on the framework must embed or have present as other bundles all its dependencies, this process is formally known has the wiring of components.

Once a bundle is deployed it passes to the ***INSTALLED*** phase, in this phase the framework wires to the bundle the required dependencies, once this process is completed it assumes the ***RESOLVED*** state, in this state the bundle can be started, if all the process goes well it reaches the ***ACTIVE*** state, being running on this last phase. It can then be stopped, backing again to the ***RESOLVED*** state, also in this state, automatically if one of the bundle dependencies was suddenly uninstalled, the bundle retreats to the ***INSTALLED*** state. Finally it can be uninstalled and remain in this correspondent state until is started again.

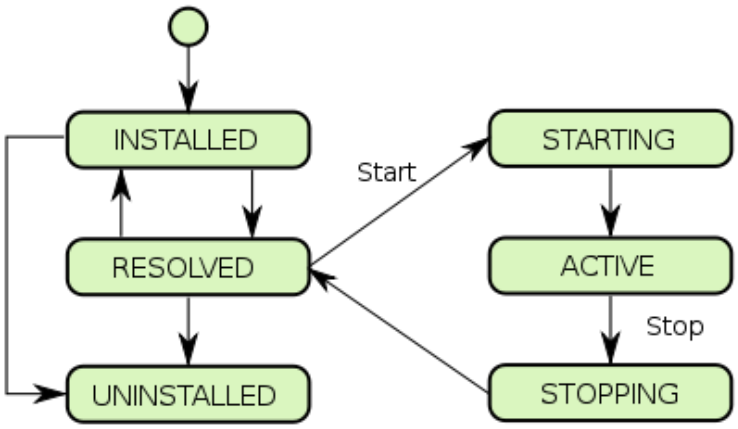


Figure 4.3: OSGi bundles lifeycle.

4.3.1.2 Apache Felix OSGi Framework

Felix framework [24] was chosen because of its extensive documentation, also including a *UPnP* specification implementation, officially integrated and developed by the *Apache Felix project*. In this work the used version of the *OSGi* core used by the Felix framework was the version 4.3. The framework provides other utility bundles that was of considerable importance for accelerating the development and for later use such administration utilities. The *Felix GoGo shell*, is an utility command interpreter bundle that allows for administration of the framework locally and remotely. Bundles are maintained in repositories, we can have several local and remote repositories, the *Bundlerepository* bundle, is another utility that comes with the framework, this utility allow for the framework to inspect local and remote bundles, it can be used combined with own developments for an application integrated bundle administration. This administration facility allows to configure connections to remote bundle repositories, this feature will be of major utility, once the architecture is deployed in a factory, when the framework is instructed to install, for example, new aggregation and validation bundles it can download those components from a central factory repository through the FTP protocol. To understand

the framework functionalities and interiorize the special considerations in developing *OSGi* applications, we have based in the book [25].

4.3.1.3 iPOJO

Being this work a modular architecture oriented to services, management logic of those services must be added to system to handle some Service Oriented Paradigm considerations. Achieve service dynamism in traditional *SOA* approaches, namely, in web service systems, is not a well adopted concept in implementation strategy. *"Indeed linking business and operation process stands to profoundly change the way application software supports business activities".* [26]

SOC approaches require enough flexibility to support dynamic coupling of service providers and consumers, thereby, the widespread concept of loosely coupling of services. *iPOJO* is a component of software, *OSGi* compliant, that is available and strongly integrated with the *Apache Felix OSGi*, it acts like a framework that covers *SOC* paradigm requirements to achieve dynamical service oriented components. Making use of this component allows to focus just in the business logic and functionality of the system components; leaving the *SOC* requirements and other non-functional mechanisms to the *iPOJO* to manage. This way the components of the system remain like *"simple old java objects", (POJO)*.

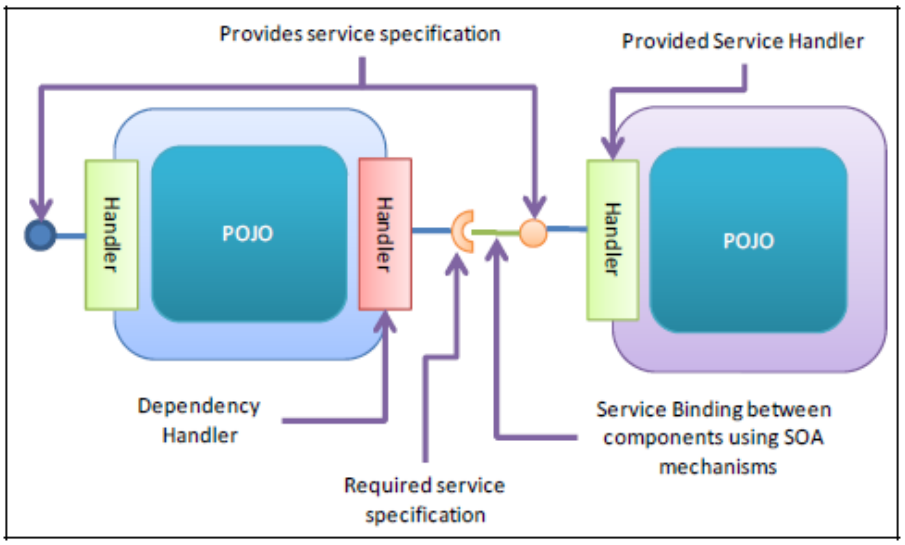


Figure 4.4: iPOJO containers and interactions.

For each component a container (like a sandbox), is created and the *POJO* representation of the component is embedded inside that container, it then manages service dependencies, publication and discovery. The container, through a *iPOJO* compliant *XML METADATA* file, creates handlers in the container to manage the POJO requirements; that file provides the service specification, properties, parameters and configuration management properties. For last, this dynamical *SOC* mechanism, manages the lifecycle of the instance; the *POJO* instance has two possible states, valid, all the component requirements are satisfied, so that, instanced is validated and can be used; invalid, some requirements are not satisfied and the correspondent *POJO* instance is not *binded*, it cannot be used. This two states are mapped respectively to *start* and *stop* callback methods, that starts or stops the execution of the system module. The *iPOJO* has assumed a crucial role in this work, because the incorporation of new system bundles, or same bundles with newer versions can be automatically managed, the architecture can be aware of components dynamic integration, therefore, performing automatically the instantiation, binding and consuming of new components, all of this without increasing the logic complexity of the overall system.

4.3.1.4 UPnP

As a stack of protocols, *UPnP* offers a set of advantages that regards a lot of the necessities exposed through the document. The necessity of interconnect devices, which can share services with the minimum effort from the users to do configurations has been the principal motivation behind this architecture protocol. The architecture was introduced by *Microsoft* with a "connected home" aim, currently is maintained by the *UPnP Forum* [4] and companies such as *Intel* have made a great effort to develop tools that we will use as helpful testers during the work. This protocol is independent from the physical platform, supports *zero configuration* in lack of *DNS* servers and promotes device connectivity allowing devices to discover each one services, subscription of events and control over devices. ***Devices***, ***Control Points*** and services are the focus of this protocol in terms of functionality purposes.

An *UPnP Device* is analogous to a server, it can embed other logic*UPnP Devices* and expose services to be invoked as well as variables that can be subscribed and queried.

UPnP Control Points are analogous to clients, they can search for *UPnP Devices* in network of their interest, subscribe to a specific device events and invoke services exposed by devices. A same physical device can contain both *UPnP Devices* and *Control Points*.

Base of the protocol are the following technologies: *TCP/IP, HTTP, HTTP* over *UDP* and *XML*. Both, devices and control points use *HTTPMU* for *UDP* multicast messages and *HTTPU* for *UDP* unicast messages. Those messages allow to control points to send *M-SEARCH* messages to discover devices in the network .From the device perspective, are used *NOTIFY* messages to announce its presence in the network. The *Simple Service Discovery Protocol (SSDP)* uses unicast messages for devices and control points to discover each ones available services and underlying service actions. The same protocol is in charge of notify device alterations *ssdp:update* messages, advertise devices leaving the network using the *ssdp:byebye* advertisement and ensure a device still connected through the *ssdp:update* message.

Once the Actions provided by a *UPnPService* are discovered, control messages to that actions can be invoked through the *Simple Object Access Protocol (SOAP)*, using the *Universal Resource Identifier (URI)* of a service the header *SOAPACTION* of a control message specifies the *UPnP Service* and the target UPnP Action in that service. *SOAP* defines the *SOAP envelope*, a *xml schema* that defines the structure of the message content, in the *UPnP* case the *envelope schema* defines input and output arguments as a field for the response to an action.

Eventing in *UPnP* is a major feature, a service can be subscribed, automatically evented variables associated to that service will notify the devices that has registered themselves interest in receive the notifications, this feature will be of remarkable importance in sensor *eventing* of data. The protocol responsible for handle this feature is the *General Event Notification Architecture (GENA)*, the messages use *HTTP* as well and *UPnP* defined templates to define the *XML* structure of the messages. Three methods are provided to handle the process, *SUBSCRIBE*, *UNSUBSCRIBE* and *NOTIFY*, a *CALLBACK* introduced *HTTP Header* is used to notify the listeners, every subscription have an identifier and a duration time, after that time the device subscribing must renew its subscription.

4.3.2 Felix UPnP Basedriver

Acting as a bridge between the framework and the network, this driver [27] allow to import and export *UPnP* devices. The *Exporter* module registers objects implementing the *UPnPDevice* interface in the framework changes in that device will be reported to the framework, the importer module, that is listening in the framework for that changes (*ServiceEvents*), will consequently export that events to the network. Events coming from the network to the exported device will follow the oposite logic path, they are notified to the framework and the correspondent *UPnPDevice* listening for service events, will be matched through its unique *ServiceProperties* and notified of external actions. The imported module of the driver acts in the same way, a *ControlPoint* announces its interest in receive *ServiceEvents* from objects *UPnPDevice*, the distinction from the devices being exported is patent in a service property that indicates that the device is imported.

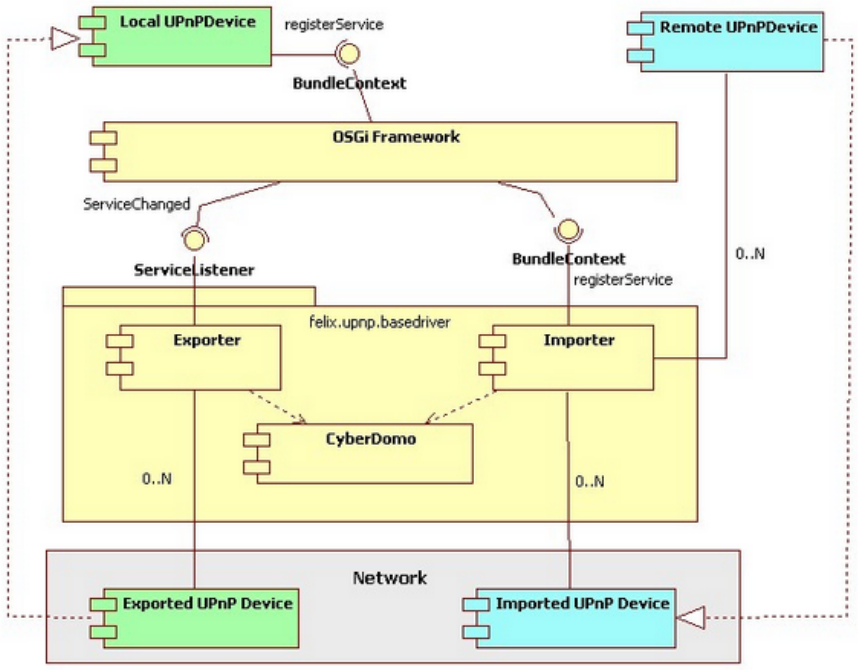


Figure 4.5: Felix UPnP basedriver overview

Chapter 5

Implementation

5.1 Contex

Project industrial partner's most challenging task is the dynamical integration of production strategies, data treatment, analyses and validation services. This challenge strictly fits the *OSGi* dynamic modularity of services, these case particular strategies can be then targeted to the system bundles. Following the system service template bundles, developers could easily, build, deploy, instantiate and dynamically manage those instances into the system.

5.2 Components

The wired set of components that figures in the figure 5.1, forms the final application, not all the components are represented, for a simplicity purpose. The components in the figure have the main responsibility of drive the architecture to its final purpose, management, reconfiguration and logistic of services. All the components implement interfaces defined in the *SmartComponent API*, the design of the final solution started by defining the whole components interface, the *Top Down* approach allowed for a clearer vision of the functionality and relation between the components. As *OSGi* bundles, every component provide a service, that service is ready to be consumed as soon every dependencies it has are resolved by the framework. If a service leaves, all consumers depending on that service must be notified and proper actions must be taken. That dependency management of the interactions between the components

requires coding to model that volatile behavior, *iPOJO* plays a preponderant role managing that *boilerplate* management, providing *callback* methods for when a service is (un)registered. Proper actions can then be taken, ensuring that the application has a much more controlled and stable behavior managing that dynamic.

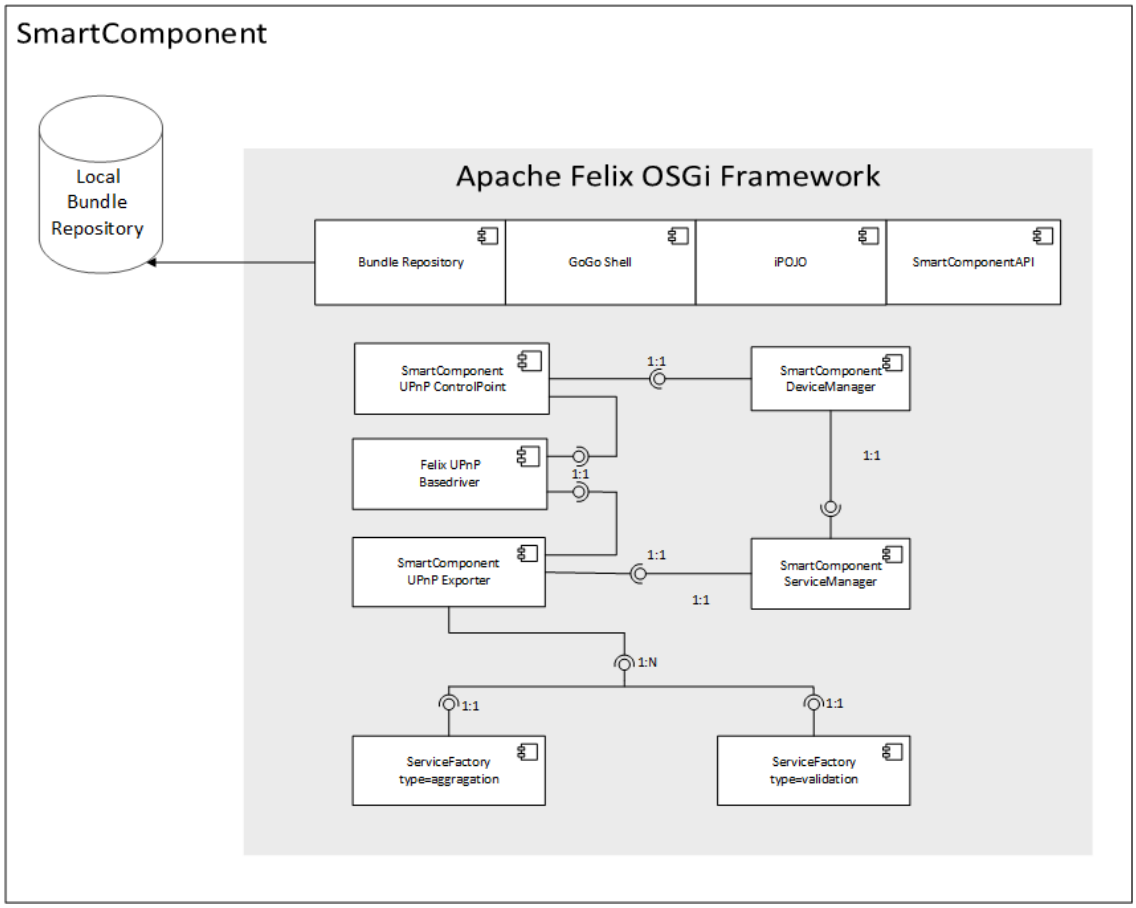


Figure 5.1: Architecture Components Diagram.

The next sections will provide an introspection of every deployed component, trying to give a deeper explanation of structural modelling and causal behavior. Additional details such as *UPnP* functionalities and *DIL* language implementation will be exposed and explained to give better understanding of the effort involved to deploy the final application.

5.2.1 DIL integration

Whereas that the language used in the *I-RAMP3* project was *DIL* and that language was target of scrutiny by the project consortium we assume that it has a solid and well defined structure that can be applied to our work. Furthermore, the context of application of the language is the inter-negotiation of services applied the Industry, that basis, also allow to put the language under the scope of the *SelSus* project.

The *JAXB* tool for allows for class model generation against a *.xsd* schema file and it was used to obtain a model class of the four *DIL* types. That approach made us choose the *JAXB* library to make the modules implement all the *DIL* related operations. One of the main efforts to integrate the language within the application, due to the dynamic nature of *OSGi*, was the use of the *JAXB* library, that has a static context. The framework was unable to resolve the dependencies of the library because it wasn't exposed as a bundle, the solution was to pack the library within each bundle that was using it and make that bundle export the library packages and import them. After that, we included the *.xsd* files and the class model of each type in a respective package, we packed that content within each bundle and finally the application was able to *(un)marshal* strings of *XML* against each *DIL* type being able to validate them. We will in the next sub chapters describe the language for each sub-type in more detail.

5.2.1.1 NSD

Tasks of *NETDEV* device are exposed to other devices through the *NSD* document, this document also defines the other three documents format (*QRD*, *TFD*, *TDD*), for that reason the documents will not be exposed. For the underlying architecture compoents that will be exported to the network, the correspondent *NSD* documents are available in the apendix section B.

5.2.2 SmartComponent API

As a central reference to the architecture, service, must define relevant properties and implement functions over that properties. As the most abstract definition, *SmartComponentService*, have four main properties:

- **Name**

  The name of the service of type *String*, is a friendly name that can represent the specific name of a complex service (eg. *MinMaxDetection*), or some property specific of the device representing it (eg. manufacturer, model).

- **Type**

  Type, in case of a sensor, represents a set of well defined physical properties that will be measured in the *Shop Floor*, in case of a complex service it could be *Aggregation, Validation, Configuration* or *Control*.

- **Version**

  A version associated to the service in the form of X.X.X, where X is an integer form 0 to 1. Version can be related to a physical in case of a device or a version of an algorithm in case of complex services.

- **UID**

  Unique Service Identifier (UID), of type *Integer*, for efficiency purpose in terms of indexation and search operations. Ahead we will show how we recurring to the uniqueness of prime numbers have implemented a strategy to retrieve that property for each service and grant uniqueness.

A complex service have providers and consumers, consequently we defined operations to add and remove them from a service. The used type to maintain the references to the associated services of a service was *ArrayList*. As previously stated in the SOA considerations, the application must regard a asynchronous processing, waiting for data from sensors to arrive, this constrain lead us to design an intermediate object between the two services data transfers, the *ServiceNode* as illustrated in the figure 5.9. The service node has a thread safe queue, when a service providing data dispatch a result to his consumers, actually is putting the data in the queue of a *ServiceNode* that is connected to the real service consumer. A service can operate in two ways:

- ***Accumulation Cycles Based***

  In this mode, a number of cycles is defined in the service, this number represents how many elements must have each provider before the data gathering is made.

To the consuming service know when he can pick the data, a number of **active** service providers is maintained, when a *ServiceNode* reaches the number of desired accumulated elements it informs the service consumer that is ready for being collected. When the total number of active *ServiceNodes* equals the number of nodes ready to be collected, the consumer service iterates the list of providers and picks the data from every one. Next the consumer resets the number of ready to being collected, processes the result and notify the result to every service consumer in the respective consumer list. For stability, when a service has an expected problem and stops sending data (eg. a device leaves the network or has a communication error), it changes its state to *IDLE* and consequently every associated *ServiceNodes* notifies the service that is consuming from its own of the new state, finally the number of active providers at the consumer is decremented by one.

- **Frequency Based**

  Based in the *Java TimerTask*, a complex service can run at a scheduled rate, given in milliseconds. Every time the service reaches the time to run the timer task *run()* method is fired, the data from the providers is collected, the result processed and the consumers are notified of the result. In this case, if a node its not active, data is not collected. What we observed in this case is that the number of collected samples for each provider is different, that difference must be handled when processing the result.

Relying on the three main components of a SOC 4.2 application and SOA architecture, a service provider is the component that provides services to be consumed, that component must register the services within the registry. The provider of the *SmartComponent* architecture is the component that scans the services available in the network, we will provide more details on the deployed component ahead. The architecture *API* reflects just the registry component and the consumer component. Lets think in the case a provider wants to register device services in the architecture, the provider must match each device property to the *SensorService*, in case of a sensor; to the *MachineService* in case of a machine. As the registry of devices, the component implementing *DeviceManager* interface, exposes himself as a service in the *OSGi* framework, and the provider knows the methods exposed by the *API*, it just

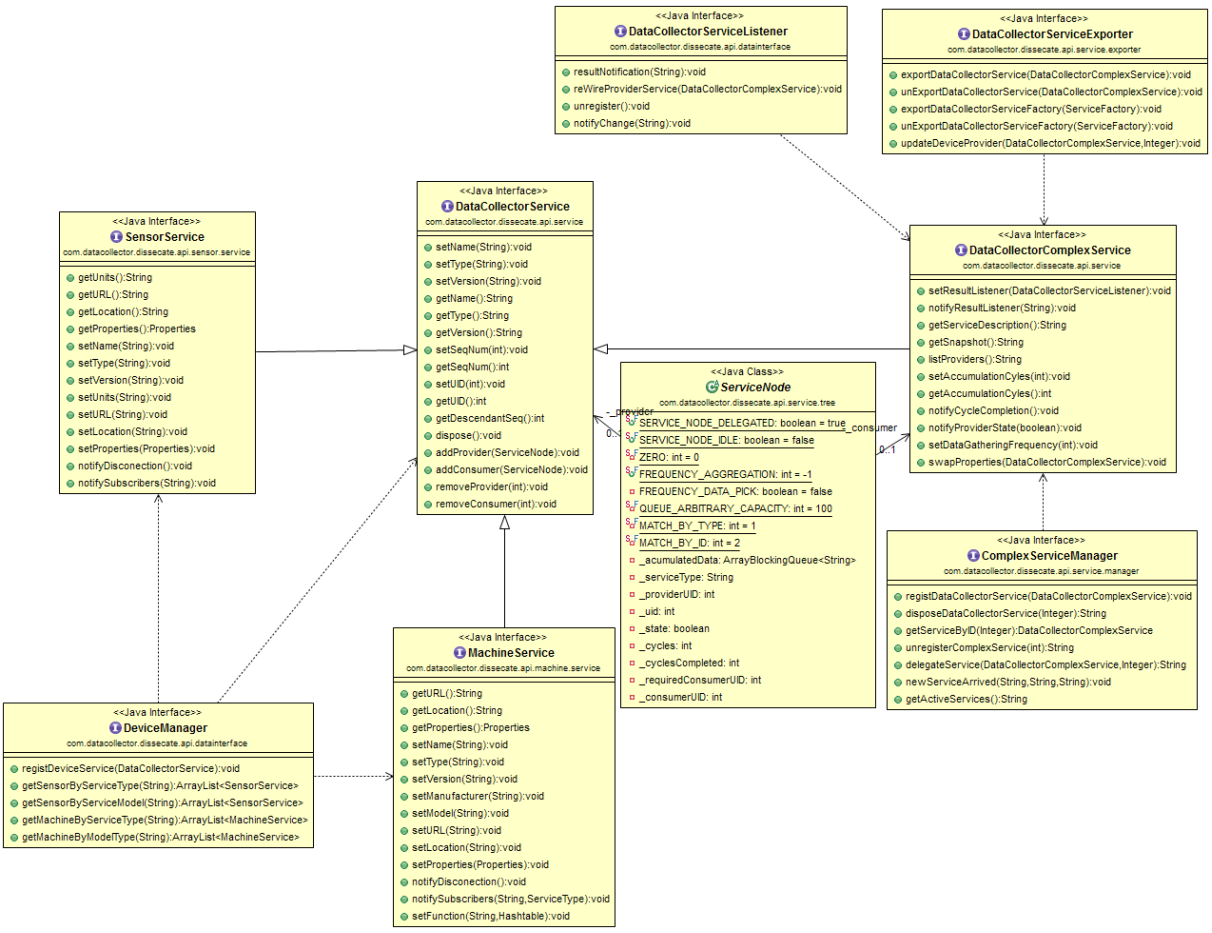


Figure 5.2: SmartComponent API Class Diagram.

needs to wire to the implementing *DeviceManager* component and use its methods. This type of interaction allows to deploy into the framework different services providers, for different protocols, we have deployed and developed a *UPnP* component, but in future developments, a Industrial protocol such *Modbus* could be used.

Dealing with two kinds of services requires, in this case, requires two registries to be present, the previous exposed is based in device services, a registry to reflect complex service needs was deployed to fill the requirements. The component implementing the *ComplexServiceManager* interface is the responsible to keep track of the complex services, those are instantiated within the architecture factory components and managed by their respective consumers. In this concrete case, the term consumer, does not have the same meaning as when we are referring to a complex service consuming another complex service. A complex service is exposed in the network, external entities

can subscribe, reconfigure and dispose that service. External entities are external consumers, they interact with the complex services through their representations in the network, a representation is reflected by the *API SmartComponentServiceListner* interface.

Every complex services registered in the implementing *ComplexServiceManager* component, are consequently exported to the network through all modules implementing the *SmartComponentServiceExporter* interface. *ComplexServiceManager* also keeps track of every exported modules registered within *OSGi* framework, making all export the complex services registered. Regarding the projects, we deployed a module that exports complex services as *UPnP* devices, again, other Industrial protocols could be implemented and used to export services.

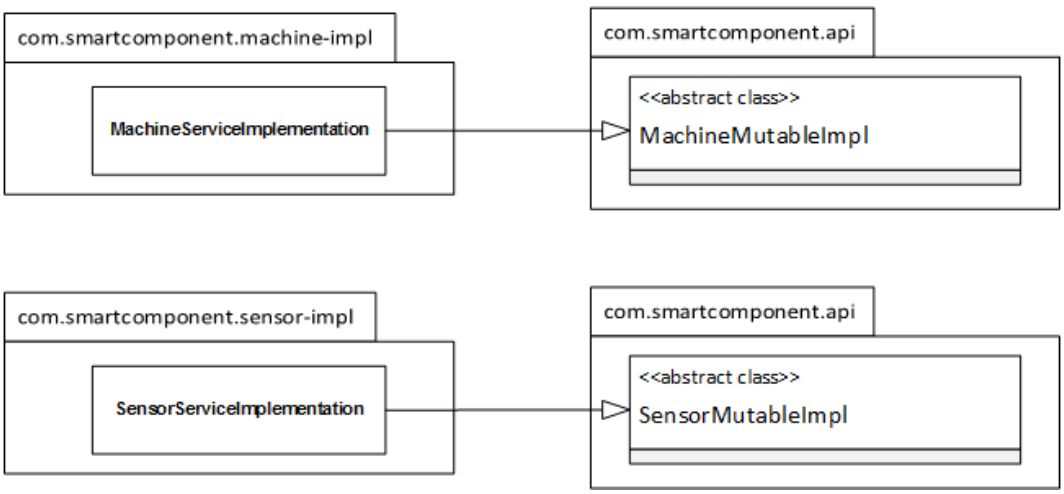


Figure 5.3: Sensor and Machine Implementation Components.

Machine and device services have different characteristics, we tried to cover that differences designing different interfaces for both, the different functionalities could be observed in the class diagram of the *API*. During the implementation we have not opportunity to test within machines, they generally have calibration and control functionalities. Sensors could have as well, but are not so complex, this exposed reflection tries to justify why we developed the concrete implementations of sensor and machine service classes as independent modules figure 5.3. As different hardware characteristics on the hardware could require new functionalities we defined a mutable abstract class implementing the sensor and machines services *API*. Defining abstract functions that

are likely to change, depending on scenario of application in terms of hardware.

All the exposed design strategies in terms of interfaces aims to explore the modular nature of the *OSGi* to make our architecture truly modular. In terms of application regarding different projects, the flexibility that make possible new component wiring and adoption of protocols, will be a valuable feature covering a larger range of requirements.

5.2.2.1 Genetic Service Identification Algorithm

Manage large sets of any type of objects, requires indexation to each element of the set. Search and retrieve a service in a registry, will be as costly as the number of services in the registry set increases. Our first impetus to map the services with a unique identifier was the use of *Universally Unique Identifiers (UUID)*, *Java* provides a library to generate that identifiers. In this explored work [28], a genetic identification of services is used, the used approach make use of simple Integer types to identify services, granting uniqueness relying on the prime numbers uniqueness. As happens in the genetics field, a gene has unique properties, that just will be reflected in its descendants. In the referenced work, a set hierarchy of sensor services was mapped recurring to a genetic unique identifier generation algorithm. The three necessary properties are, a list of **n** (n based on the expected number of services) pre computed prime numbers, a sequence number that identifies the order in which a service was inserted in any hierarchy leaf as a new leaf and the parent leaf *uid*. As the comparison of *Integer* types is more efficient than *UUID* comparison, we decided to adopt that approach with a slightly change to embrace the *SensorNode* strategy.

A mathematical explanation of the process is detailed below, the *Gene* is the *UID* of the service, the sequence number is the number of order in which the service is created (denoted in the expression below by "i") and the prime number is the prime of the pre computed list of index "i". We added an additional part to encode the descendants of a service, the *ServiceNodes* from where the service consumes other services. For a proof of uniqueness we developed a script in *Python* that show that as for service encoding as for the nodes encoding in a pre-computed list of 10000 primes every number was unique.

$$S \rightarrow SmartComponentService$$

$$X \rightarrow Set\ of\ SmartComponentServices$$

$$P \rightarrow Set\ of\ prime\ numbers$$

$$g_S \rightarrow Gene\ of\ SmartComponentService$$

$$g_{S_i} = S_i \in X \cdot P_i \cdot i : \{1 < i \leq |X|\}$$

$$Sn \rightarrow SensorNode$$

$$g_{Sn} \rightarrow Gene\ of\ ServiceNode$$

$$X_{S_i} \rightarrow Set\ of\ S_i\ predecessor\ ServicerNodes$$

$$g_{Sn_i} = Sn_i \in X_{S_i} \cdot P_i \cdot i : \{1 < i \leq |X_{S_i}|\}$$

Applying this strategy we expect to obtain more efficiency in search and management of services, another advantage, for a given node, if we divide its *UID* by its sequence number and by its prime number, we can deduce easily the parent of that node.

5.2.3 SmartComponent Device Management

This bundle keeps track of the device services, registered within it, by the applications modules that scans device services in the network. The class *DeviceManagerServices* exposes in the *OSGi* framework this component logic as a service. Automatically, all bundles interested in register device services can use that class interface to register the services, by that time already converted in *SmartComponentService* objects.

The iPOJO file image below, denotes the properties identify this component within the framework, a bundle declaring an import dependency in this service, when deployed in the framework will be automatically injected with *DeviceManagerServices singleton* instance that exposes the logic of this service to other bundles. The *DeviceManager* class handles the registration and un-registration of services as search and retrieval

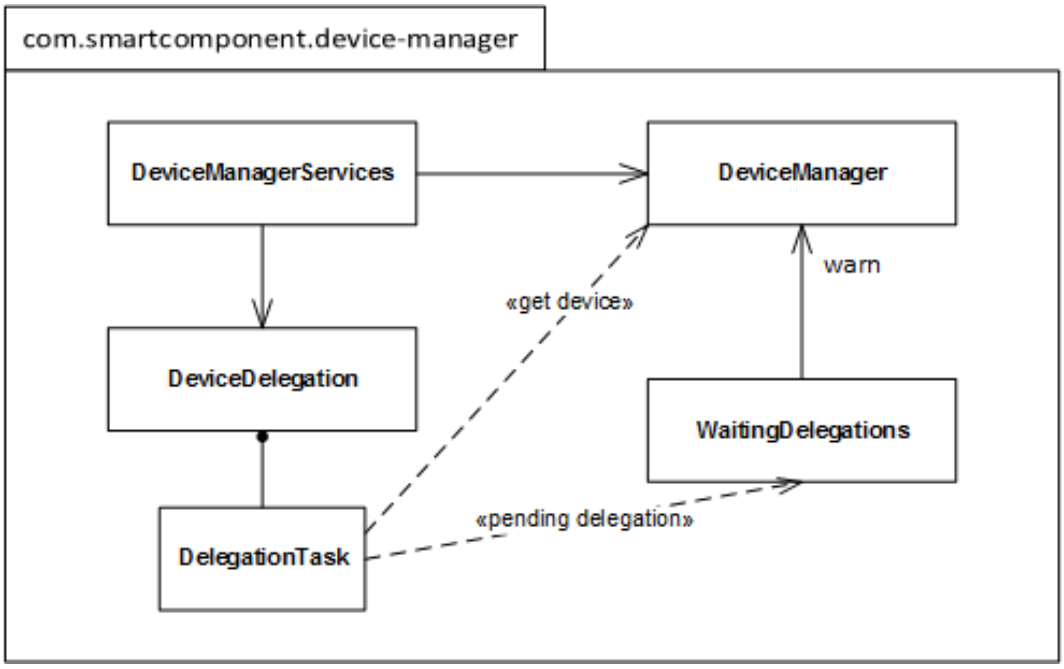


Figure 5.4: DeviceManager Class Diagram.

methods for those. Once a service instance is registered, the *WaitingDelegations* class, that keeps delegation tasks that have no match in the registry, verifies if the new service properties are coincident with the required in the task, and puts the new service as provider of the service waiting for the delegation. A delegation task is an object that contains the reference to a *IDLE* state *ServiceNode*, that for its turn, contains reference to a consumer complex service that regists its interest in consume a service of a determined type (eg. an aggregation service wants to consume from a specific Temperature service). The complex service interest is submitted to the *DeviceDelegation* class, that searches in the registry for a compatible service, if that compatible service does not exist, then, the delegation task is inserted in the *WaintingDelegation* class. A *DelegationTask* is literally a task in the logic perspective as it is consumed by a thread, and that tasks are maintained in a thread safe queue at the *DeviceDelegation* class instance. When a component providing device services leave the framerwork, the *iPOJO start()* and *stop()* callback methods allow for before the bundle leaves, it unregister all its provided services from the registry.

```
<ipojo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="org.apache.felix.ipojo
      http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd"
    xmlns="org.apache.felix.ipojo">

    <component
        classname="com.smartcomponent.device.management.DeviceServices"
        name="DeviceServices" public="false" immediate="true" architecture="true">
        <provides />
        <provides
            post-unregistration="unregistered" post-registration="registered"/>

    </component>
    <instance component="DeviceServices" name="smartcomponent.device.management.deviceservices" />
</ipojo>
```

Figure 5.5: iPOJO DeviceManager metadata file.

5.2.4 SmartComponent UPnP Control Point

SmartComponent, will announce all the sensing devices as *UPnP* devices, encapsulated as *NETDEVs*. From this operation scenario, this component has emerged, representing a *UPnP Control Point* that will scan the network for all devices of ***UPnP Device Type : urn:schemas-upnp-org:device:NETDEV***. More specifically, the *Felix UPnP Basedriver* is the component that interacts with the devices, when it finds a device, fires a *OSGi ServiceEvent*, as the *RootDeviceListener* class registers within the framework interest in receive *UPnPDevice* events, receives the event and if the device is of the type refereed previously proceeds the execution of the logic exposed in the sequence diagram 5.2.4. Three other omited cases in the diagram could happen, device does not export the ***UPnP Service Type : urn:upnp-org:serviceId:NetdevService***, that device is discarded; validation of the *NSD* document against the schema is not valid, device is discarded; a specific device and the associated sensor already exists in the registry, device is discarded and a warning message is logged because this can represent a potential problem of hardware.

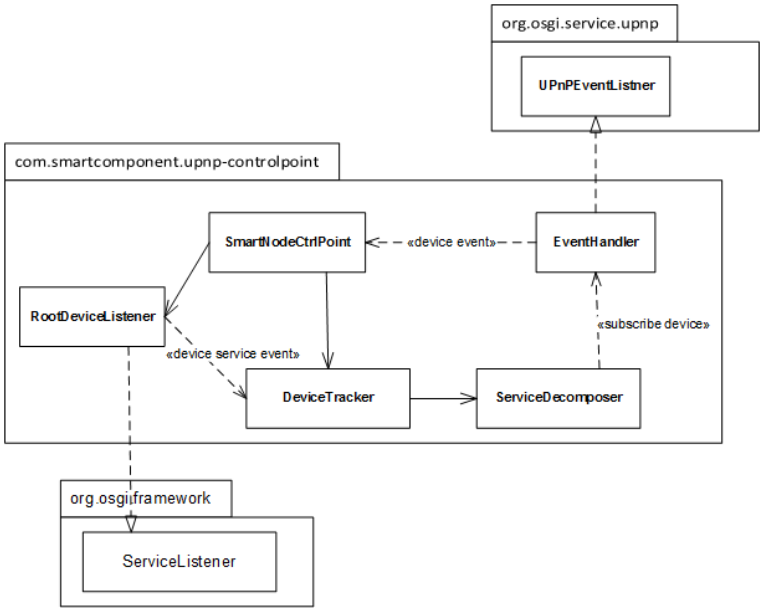


Figure 5.6: SmartComponent ControlPoint Class Diagram.

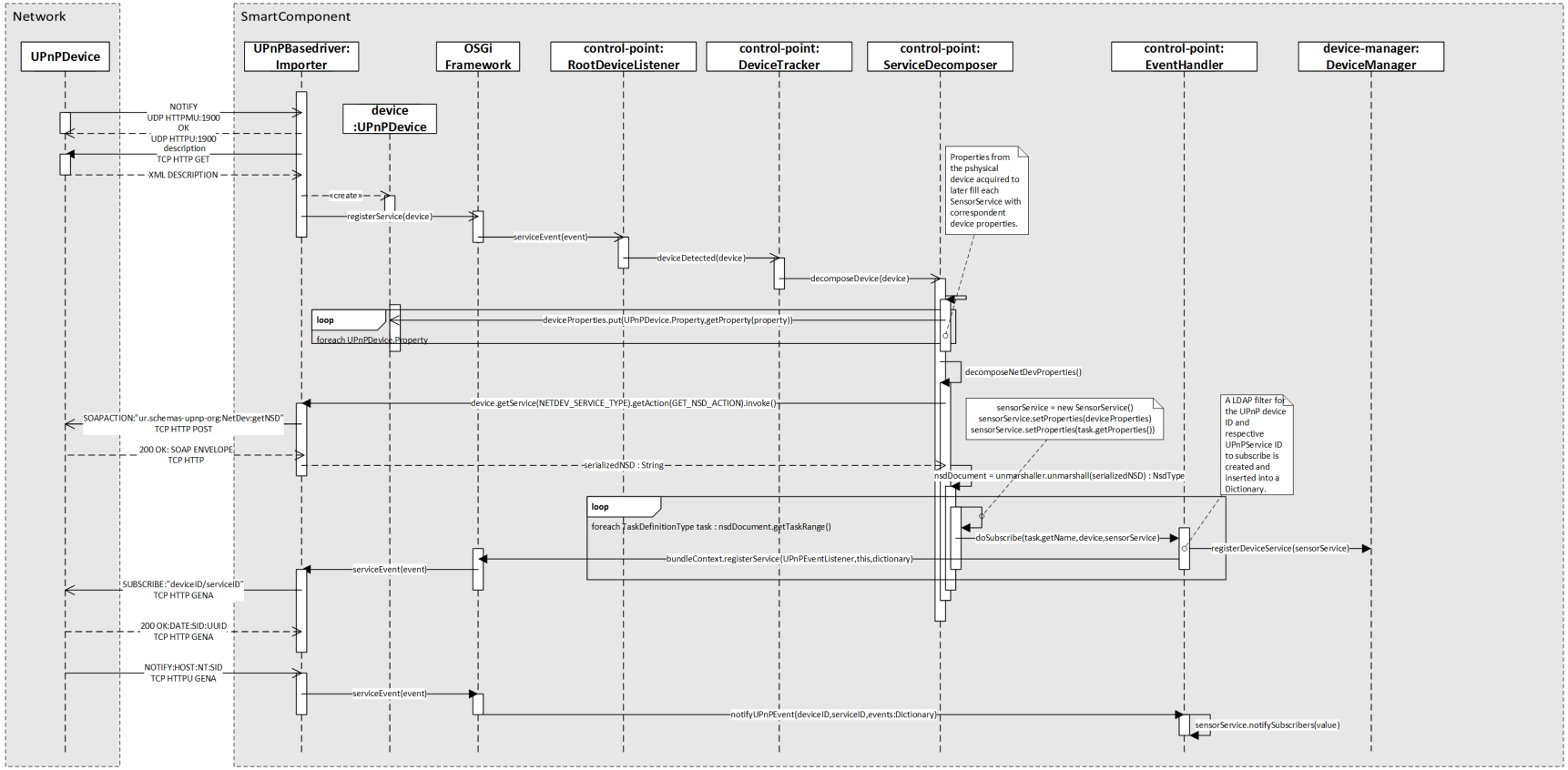


Figure 5.7: Sequence diagram service subscription.

5.2.5 SmartComponent Service Manager

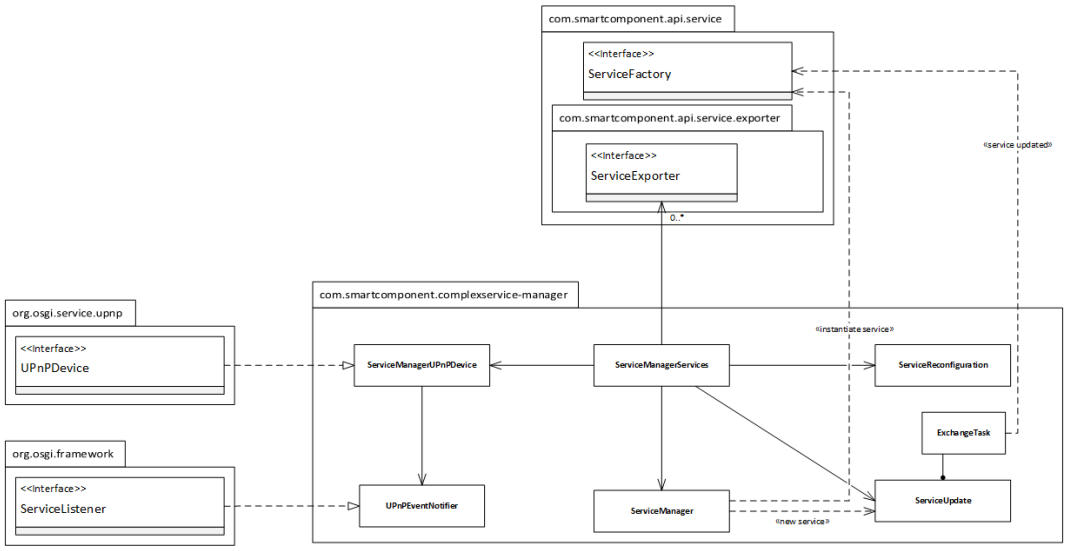


Figure 5.8: ServiceManager Class Diagram.

ServiceManagerServices class exposes services to framework, beyond that, *ServiceFactory* and *ServiceExporter* modules of the architecture must register within this component. Complex services management is handled in this component, as registry and reconfiguration operations, *ServiceManager* class handles the registry operations, keeping track, removing and retrieving all the registered complex services. A reconfiguration of a service is categorized in two main operations:

- **Service wiring operations**

  To a service, providers and consumers can be assigned or removed. When required to do such an operation of reconfiguration, network representation of a concrete service use the *ServiceReconfiguration* class to notify that required changes. Those can be the assignment or removal of a device service or a complex service, in case of a complex service the match to the required instance is done at the *ServiceManager* class; in case of a device service *DeviceManagerServices* are used. For a better understanding of this reconfiguration, image 5.9 could provide a better understandable of how the reconfiguration process wires together different services, using *ServiceNodes* as intermediate connectors, removing from the service itself the complexity of managing the active connections

and reconfiguration of new ones.

- **Service update operations**

  Specific algorithm implementations of aggregation and validation services are kept in that respective factories. When a newer algorithm version is deployed into the framework, the correspondent factory announces to the *ServiceManager* class that new service version. If the new model version is newer that the previous, and service instances exists of the previous versions, a instance from the newer version module is created through the correspondent factory. That instance is passed to the *ServiceUpdate* class that schedules a correspondent *ExchangeTask*. Scheduled tasks have reference to the older and newer instances, the objective is in runtime exchange of service versions, without the need to disconnect that service representation of the network, this way, not breaking connections to other external entities consuming the services through their network representations. The UPnPDevice exported can maintain the established connections with external actors, while this, the complex service performing the tasks is exchanged for annother of the same type, same name, but different logic as it is a newer version.

Once a complex service is registered in the *ServiceManager*, that instance is announced to all the *ServiceExporter* modules registered. Different protocols could be implemented by the exporter components, this design approach regards future requirements off different protocols to be used, thus, enhancing the architecture flexibility to adapt to different application environments. We developed *SmartComponent UPnPExporter* component that currently exports the complex services as *UPnPDevices*, encapsulated by a *NETDEV* entity.

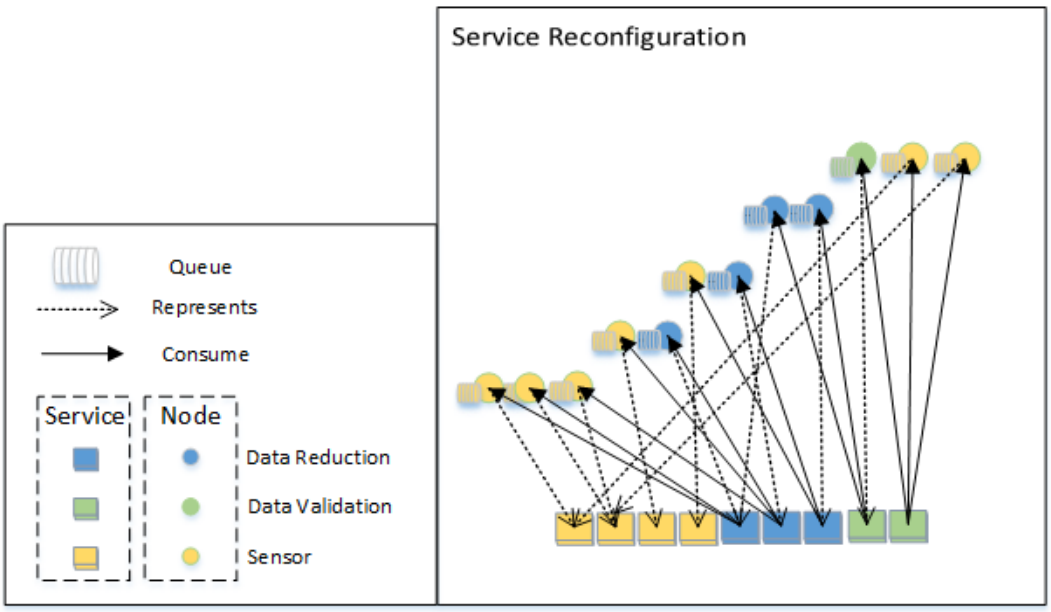


Figure 5.9: Service Wiring Structure.

5.2.5.1 UPnP exposed funcionalities

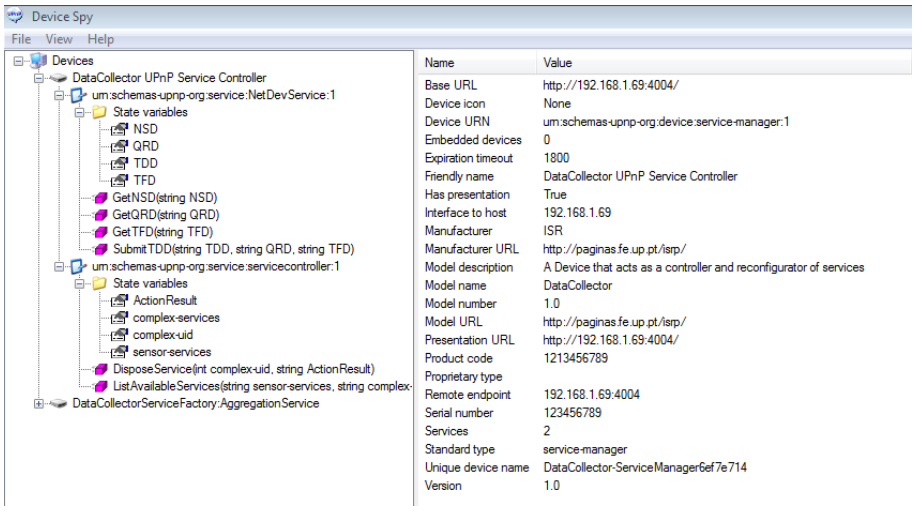


Figure 5.10: ServiceManagement UPnP Device.

Operations over this component are accessible through a *UPnPDevice* in the network. As we can observe in the figure 5.10 above, two *UPnP Services* are exposed, the *urn:schemas-upnp-org:service:NetDevService:1* perform exactly the same functions as the exposed *urn:schemas-upnp-org:service:servicecontroller:1*. They are announced

in the *NSD* document and can be invoked submitting a valid *TDD* to this device that will reply with *QRD* to a correct invocation and with a *TFD* document otherwise. The actions in the *urn:schemas-upnp-org:service:servicecontroller:1* service will produce the following:

- ***DisposeService(int complex-uid, string ActionResult)***

  For the input argument, *complex-uid*, the correspondent complex service with the same *UID* will be disposed and correspondent *UPnPDevice* network representation in the network will disappear. An opposite action such *InstantiateComplexService()* is not present in this *UPnPService* since the factories that produce instances export that functionality and provide information about correspondent modules that they can instantiate.

- ***ListAvailableServices(string sensor-services, string complex-services)***

  The two seen arguments are output arguments and this invocation produce a list off all the active instances of sensor and complex services running at the time of invocation.

5.2.6 SmartComponent UPnP Exporter

For each registered complex service and *ServiceFactory*, exists the need to a correspondent network representation, that is done by this component. Using its *BundleContext* the *UPnPDeviceManager* class regists in into the framework each service and factory correspondent *UPnPDevice* representations. The underlying logic of the representations is occulted for simplicity purposes, actions exposed by that devices will be explained ahead. *UPnPDevices* that represent complex services implements the *SmartComponentServiceListner* interface, this way, the complex service does not need to be aware of what kind of object is representing them, this interface has methods for result notification of the services, alterations in terms of providers and consumers and a rewiring method in the case the service is updated. In this case a *thread safe lock* mechanism is used for prevent calls from the network representation to the service behind while the change is being performed. Factory devices are managed by the same class, the registered factories itself are managed by the *UPnPFactoryManager*, that make each factory being exported or removed from the network, as the bundles representing the factories come and leave the framework.

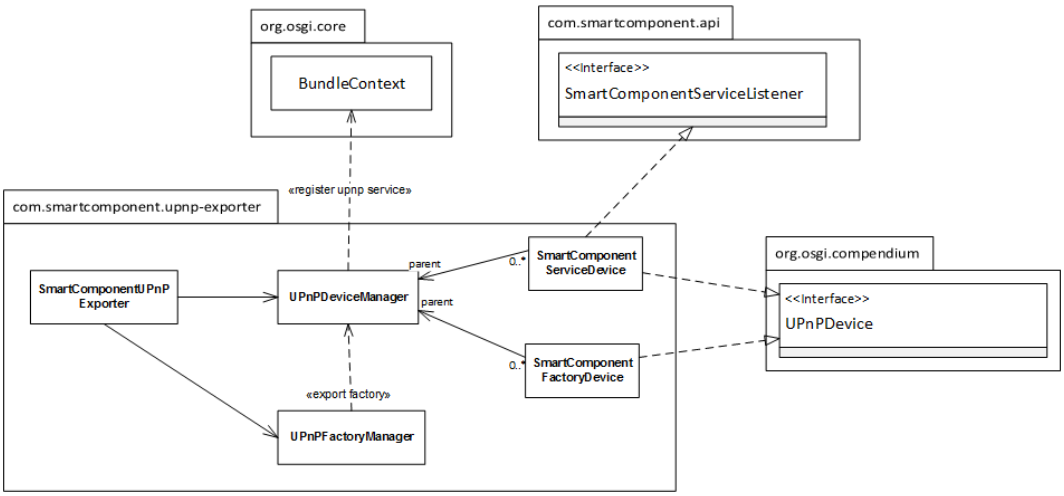


Figure 5.11: SmartComponent UPnP Exporter Class Diagram.

Metadata file for this component to be anounced in the framework is exposed above.

```
<ipojo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="org.apache.felix.ipojo
      http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd"
    xmlns="org.apache.felix.ipojo">

    <component
        classname="com.smartcomponent.upnp.exporter.SmartComponentUPnPExporter"
        name="SmartComponentUPnPExporter" public="false" immediate="true" architecture="true">

        <provides
            post-unregistration="unregistered" post-registration="registered"/>
        <requires field="serviceManager"/>
        <callback transition="validate" method="start"/>
        <callback transition="invalidate" method="close"/>

    </component>

    <instance component="SmartComponentUPnPExporter" name="smartcomponent.upnp.exporter.smartcomponent.upnpexporter" />
</ipojo>
```

Figure 5.12: iPOJO metadata file SmartComponentUPnPExporter.

5.2.7 SmartComponent Service Factories

Service factory components aggregate all the modules that will produce complex services by type. As before said, a type can be *Aggregation*, *Validation*, *Configuration* and *Control*. Our developments have focused in *Aggregation* and *Validation*, however, the architecture design adopts ways to create and deploy the other two, or newer factory types. A factory is a component that will aggregate all models of its type, a model in the framework perspective is a bundle that will be deployed in the framework. As the determined factory type produces instances of the available bundles of that type, it must know the bundles that encapsulate models of the same scope. To do this we have recured to the *WhiteBoard Inversion of Control* pattern, the bundle must declare in its own *manifest.xml* file the type it represents, the factory in its turn, must declare itself as *iPOJO Extender* and declare callback methods for treat the bundles arriving or leaving the framework, this process is patent in the figure 5.14, the *metadata.xml iPOJO* file for the aggregation factory.

```
<ipojo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="org.apache.felix.ipojo http://felix.apache.org/ipojo/schemas/CURRENT/core.xsd
      org.apache.felix.ipojo.extender http://felix.apache.org/ipojo/schemas/CURRENT/extender-pattern.xsd"
  xmlns="org.apache.felix.ipojo"
  xmlns:extender="org.apache.felix.ipojo.extender">
    <component
        classname="com.smartcomponent.aggregation.impl.AggregationServiceFactory"
        name = "AggregationServiceFactory" public="false">
        <extender:extender
            extension="Aggregation-Service" onArrival="onBundleArrival" onDeparture="onBundleDeparture" />
        <callback transition="invalidate" method="stop" />
        <callback transition="validate" method="start" />
        <requires field="servManager"/>
    </component>
    <instance
        name="aggregation.service.factory"
        component="AggregationServiceFactory">
    </instance>
</ipojo>
```

Figure 5.13: iPOJO ServiceFactory metadata file.

The following flow diagram details the process involved in recognise a bundle extension type and association to the correspondent extender (factory):

1 - The aggregation bundle is deployed, activated and started within the framework.

2 - The framework triggers a bundle *STARTED* event, registered as a listener the iPOJO Extender catches the event.

3 - The *iPOJO Extender* inspects the bundle *manifest* file headers, consequently processes its components.

4 - The *iPOJO Extender* registers the aggregation specific module service in the framework.

5 - Next, the framework triggers a service *REGISTERED* event of that bundle, listening that event is the *AggregationFactory Whiteboard Extender*.

6 - The *AggregationFactory* inspects the classes of the service and verify that it extends the *AggregationService* class.

7 - The *AggregationFactory* informs the *ServiceManager* of the new module and it will proceed to a verification in the active services registry.

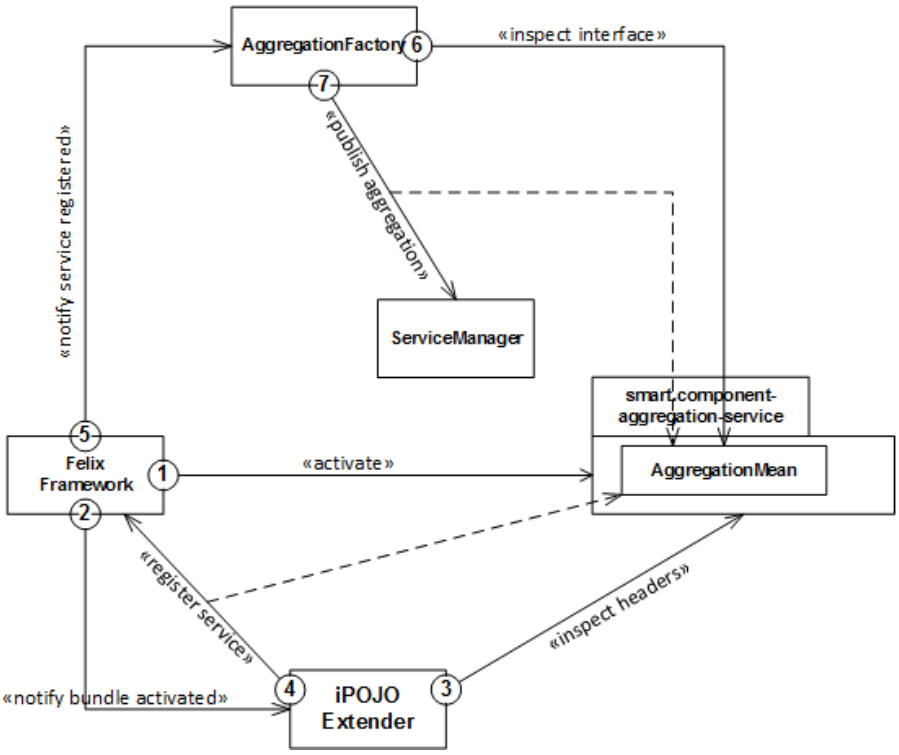


Figure 5.14: Extender Witheboard Pattern.

5.2.7.1 UPnP exposed funcionalities

As *UPnPDevices* the factories expose to other actors the following functionalities, the image below ilustrates the *UPnPDevice*, the example given is for the *AggregationFactory*:

- **InstantiateServiceAccumulationCycles(string ServiceID, string AccumulationCycles, string sensor-type, string sensor-uid, complex-type, complex-uid, string ActionResult)**

  This action creates a new complex service, the first argument specifies the name and version of the model to instantiate (eg: *Mean:1.1.0*); second argument specifies the size of the data set that will be collected from each provider; third argument specifies the types and numbers of each type of services that will be consumed for the service (eg: TemperatureService:2,LuminosityService:4); fourth argument specifies specific sensor services that will be consumed by its *UIDs*; fifth argument specifies the types and numbers of each type of complex services

to consume (eg: *Mean:1.1.0:2,SimpleSum:1.2.0:4*); sixth argument indicates the specific complex services that will be consumed by its *UIDs*; last argument is the output argument that returns the result of the instantiation. It indicates the lack of specific service if that service does not exist and that a indicated *ServiceID* does not exist if the factory does not provide such module.

- **InstantiateServiceFrequencyData(string ServiceID, string GatheringFrequency, string sensor-type, string sensor-uid, complex-type, complex-uid, string ActionResult)**

  Only one difference exists between this and the previous action, the parameter *GatheringFrequency* indicates in *milliseconds* the interval in with the data from providers will be collected.

- **ListAvailableServices(string OperationResult)**

  OperationResult is an output argument that in this action will indicate the modules that this factory can instantiate.

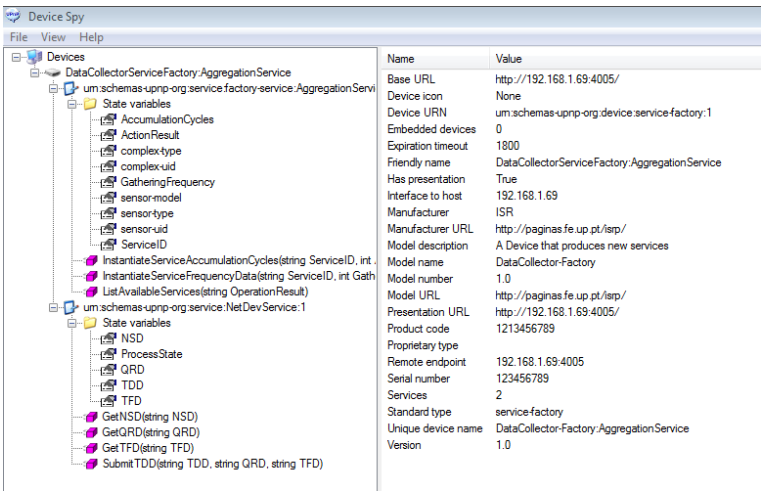


Figure 5.15: ServiceFactory UPnP Device.

5.2.8 Data Aggregation and Validation Services

Implementation of aggregation and validation complex services was designed with flexibility to adapt the way the data from services being consumed is handled. The classes responsible for handle that management are the *ValidationService* and *Aggregation-*

Services abstract classes, in the first case, when analysing data from the providers, a validation of the data is being done, that requires to get more detailed information from the providers. In case of detecting an anomaly, the result of a validation service must indicate the service, or the services, that have malfunctions with detailed information than allow to identify the source. In the second case, we have developed simple mean and sum methods, in that case the data is simply aggregated and do not require specific information, consequently the *ValidationService* abstract class does not have a complex logic as the *ValidationService*.

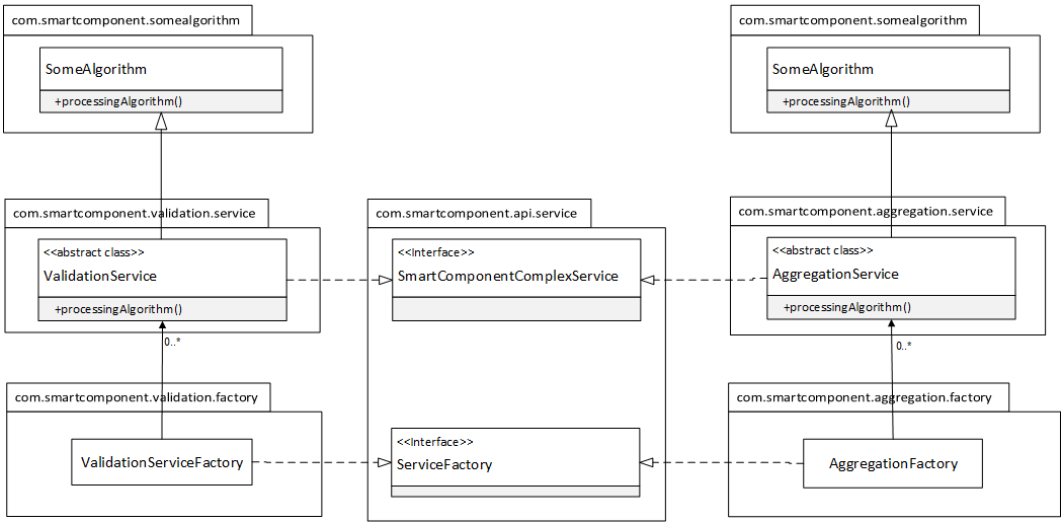


Figure 5.16: ConcreteComplexServices Class Diagram.

The concrete implementation of a service is illustrated by the classes *SomeAlgorithm* as figures in the class diagram figure 5.16. Overriding the correspondent service abstract classes, the concrete classes must implement the method *processingAlgorithm()*, the input arguments of that method are omitted because of the stated before, the input will vary according to the necessity of more complex logic, as happens in the validation services. This approach allow to adapt the implementation to new required functionalities as the granularity of components allows to deploy more complex implementations of underlying components as needed. When the result of an algorithm is processed the concrete implementation class must invoque the callback method *algorithmResultReady(result)*, at the *Super Class* to trigger other methods that will notify the network representations of that service implementing the interface *SmartComponentComplexServiceListener*.

Packaging those models as bundles, as they are extensions of their respective *Servicefactory* types, requires to declare in the *manifest.xml* file of the bundle, that they are extensions. This way when deployed in the framewrok the *WhiteBoead* pattern will recognise those bundles as extensions of a factory and deliver the notifications that will match them their correspondent listeners, this property is ilustrated in the *pom.xml (project object model)* file of an aggregation bundle 5.17.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.smartcomponent</groupId>
    <artifactId>com.smartcomponent.aggregation-mean</artifactId>
    <version>1.2.0</version>
    <packaging>bundle</packaging>
    <name>Reduction Service Simple Mean</name>
    <description>Implementation of a simple mean reduction service</description>
    <dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <version>2.1.0</version>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        <Bundle-Category>base</Bundle-Category>
                        <Aggregation-Service>com.smartcomponent.aggregation.mean.MeanAggregation</Aggregation-Service>
                        <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
                        <Export-Package>com.smartcomponent.aggregation.mean;version="1.0.0"</Export-Package>
                    </instructions>
                    <remoteOBR>repo-rel</remoteOBR>
                    <prefixUrl>file://C:\thesis.project\smartcomponent\releases</prefixUrl>
                </configuration>
            </plugin>
            <plugin>
            <plugin>
        </plugins>
    </build>
```

Figure 5.17: iPOJO ComplexService POM file.

5.2.8.1 UPnP exposed funcionalities

SmartComponentComplexService send periodical results, an external entity that is interested in subscribe the results, must subsribe the *UPnPService urn.schemas-upnp-org:service:AggregationService:1*, this service provides an *UPnPEventedVariable* that can be susbsribed and is actualized by the underlying complex service sequential results. The *UPnP* protocol allows for an easy subscription of events relying in the *GENA* protocol. In case of *NETDEVs* communication, the subscription of the correspondent figuring service, will subscribe the *QRD* variable, this variable is also evented and sends the service result embedded in that document.

- ***AddConsumer(int complex-uid, string ActionResult)***

For the input argument *complex-uid*, the correspondent complex service with matching *UID* will be associated as a consumer of the service that represents the action. The result of the operation, in case of success or failure comes in the *ActionResult* variable.

- ***AddProvider(int complex-uid, int sensor-uid, string ActionResult)***

  For the input argument *complex-uid*, the correspondent complex service with matching *UID* will be associated as a provider of the service that represents the action. The result of the operation, in case of success or failure comes in the *ActionResult* variable.

- ***GetLastResult(string Result, string ActionResult)***

  The two figured variable are outputs, the *Result* variable is the evented variable that handles each new result of the underlying complex service. This action will retrieve the last result of that variable.

- ***GetSnapshotResult(string Result, string ActionResult)***

  In case an instance value of the service is needed, this action must be called, data from the wired *ServiceNodes* will be picked, and an instant result will be processed and retrieved in the evented *Result* variable. If some error occurs that error will be notifyed in the *ActionResult* variable.

- ***ListProviders(string ActionResult)***

  All services providing that to the correspondent service from where this method is called are listed, separated as device services and complex services.

- ***RemoveProvider(int complex-uid, int sensor-uid, string ActionResult)***

  For a matching device or complex service, if that service is being consumed by the service associated to the action, that correspondent provider will be removed.

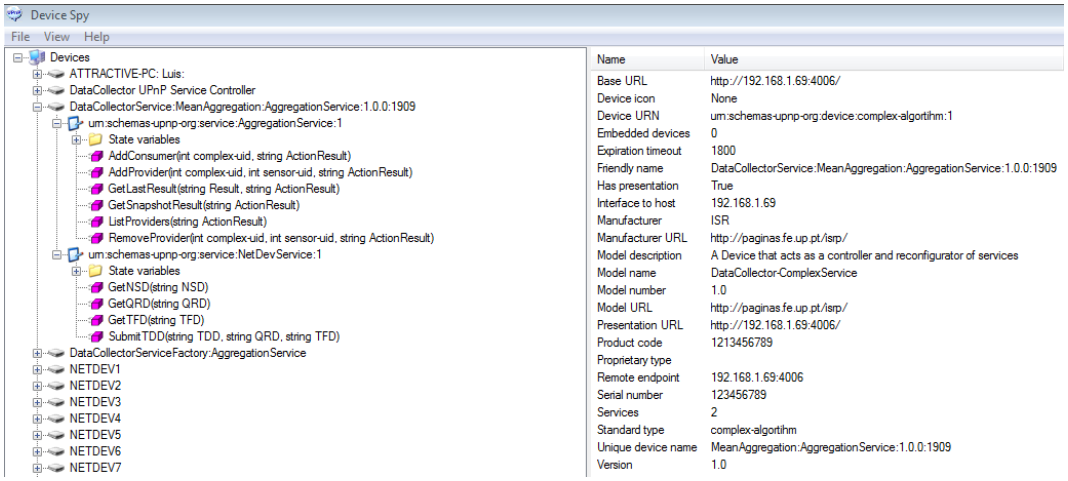


Figure 5.18: Complex Service Exported Device.

Chapter 6

Case Study

6.1 Introduction

The context of the hypothesis is based in one of the projects Industrial partners, providing a realistic context of application to the case study, due to confidentially questions we will not reveal the specific partner.

Assembling engines is the main activity of one factory that belongs to that partner. One of the sub-tasks is the sealing of parts that are assembled together to avoid oil leaks. The oil leaks are prevented by applying a sealant in the contact surface of the two parts, a special sealant is used in the factory. The application process requires metrics from a crucial set of the surrounding environment conditions, which must be gathered under strict parameters. Below are the conditions that must be monitored during the process, with detailed explanation of the process. Humidity in the air, temperature in the zone of application, pressure in the process of join engine parts and optical sensors to verify if the surface of contact has imperfections. The sealant is applied by a complex machine, a *Robotic Arm*, the optic sensor will be installed in the tip of the arm. During the application of the product, imperfections are detected, preventing the defective part to continue through the production stages before the defect is corrected. Near the robotic arms other types of sensors will be placed; temperature and humidity sensors, monitoring the surrounding environment where the process is occurring. That conditions can affect the drying time and the hardness of the sealant, drastically reducing its lifetime. The pressure sensor is coupled to the arm, its function

is to measure the pressure between the contact surfaces of the parts. Ensuring that the sealant is uniformly distributed over the contact surface, requires that a correct level of pressure is applied.

Another complex machine that needs to be monitored in this scenario, it is known as *NC-Axis* and its function is to do the transportation of parts between sections of assembly. These machines are subjected to stress when loading excessive weight, causing the premature degradation of its mechanical components, such as motors, gears and the premature change of belts that are used to coordinate the machine shafts. Due to nonlinear weight transportation, the duration of the belts tend to be unpredictable, leading to cases in which the belts break, and consequently, the machine needs to be stopped, interrupting a whole sub-section of the production process. Vibration and pressure sensors need to be installed in these machines, these will allow to predict the degradation of the parts. Knowing the approximate or the total number of times a machine has been loaded in overweight, preventive maintenance can be performed avoiding major consequences.

Regarding the above exposed situations we will form and present a hypotheses that will be the base to validate our expected results.

6.2 Hypothesis

Our formulated hypothesis will focus on the previous scenario of application, we will predict a system behaviour for each one specific production contexts:

- **Component**

  A component is to be intended as a single production equipment, regarding the previous sub-section we will analyze two components, the *Robotic Arm* and the *NC-Axis*. Our assumptions are that for single components the system will respond with a normal behavior, to the analyses of **4 sensors** in arm case and **2 sensors** in the case of the second machine. By normal behavior, we refer to characteristics that makes possible to use of the system under real time needs, frequencies of operation in the second's domain. The set of sensors that monitor each machine, will be respectively connected to a mote and each mote placed near each machine. In this case, the analyses will be performed by the developed *MinMax*

module. A validation service that fires warnings for measures that exceeds min and max thresholds for each specific type of service.

- **Equipment**

  Combinations of two components forms equipments, an equipment is a single station or sub-process of a *Supply Chain*. When a robotic arm finishes the process of applying the sealant, the engine is transported in the *NC-Axis* to other sub-process of production, the combination of these two sub-tasks forms a main task. This previous process forms another hypothesis, **6 sensors** (4 in *RTV arm's* and 2 in*NC-Axis*), must be analyzed to control the main production task efficiency. Again, we will use the *MinMax* module for anomaly detection over each equipment. For storage efficiency and other possible necessities, we will deploy modules *MeanAggregation* and *DummySum*. The first one does a simple mean of the given sensors input and the second produces a sum of all the inputs. A service for each sensor service type will be instantiated, aggregating all the sensors in the network of the same type, in the same *virtual sensor group*. Comportment of the system is expected to be normal, for aggregation services, an output frequency of **4 seconds** will be required. In validation service a frequency of frequency of **5 seconds** will be used.

- **Supply Chain**

  Equipments, performing a same task over a specific physical area in the factory, resulting in output, that feed another main task, form a *Supply Chain*. Focusing in the concrete case of the partner, a *Supply Chain* of several previously exposed equipments have generally between **10 - 100 sensors**. Regarding the previous interval, a maximum number of sensors, represents the presence of **16 equipments** in a *Supply Chain*. To validate this situation, we will scale the previous hypothesis solution, for each iteration over the number of equipment's, adding a new validation service and aggregating the additional sensors that will be present. We expect an increase in the time that takes to detect the sensors exposed in the network, a problem of the protocol itself, due to the considerable volume of packets flowing in the network. Excessive packets are originated by *advertisement*, *subscription* and *control* messages that are used by the *UPnP*

protocol. Regarding the architecture itself, we will base in the normal behaviour of aggregation and validation services as proof that a possible exponential growth in detection of sensors, is due to the protocol. If locally the services are detectable, and remotely they are not visible, we will be facing network and protocol issues, rather than architecture issues. We will compare the detection against the remote machine using the *UPnP Device Spy Tool*, a control point designed by *Intel* to test *UPnP Devices*.

- **Shop Floor**

  Regarding the *Shop Floor* of a factory, several *Supply Chains* are deployed all over the area of production. Regarding the Industrial partner case, we will point a minimum number of sensors, based in the mentioned number at one single *Supply Chain*. In a *Shop Floor*, a minimum of one and a maximum of six *Supply Chains* are present, this gives us a test interval of **100 - 600 sensors** in all the area of production. Once again, one of the expected problems is the device detection delay. Response time will be the proof used to validate the architecture normal function against protocol problems. In this case a range of **16 - 96 complex services** will be instantiated and its response times and expected behaviors evaluated. Due to the high number of devices in the network, we expect at some point, an exponential growth of the times that takes to detect devices and invoke actions.

A tool that simulates *UPnP NETDEV* device instances in the network will be used, that tool creates the required number of devices with different service types. It allows to define the mean and variance of each service, this way, creating purposeful failures that must be detected by validation services. The sampling rate of each simulated sensor will be of **1 second**. The types used will be the previous stated types of sensors, that simulation tool will be running in a different machine than the *SmartComponent* architecture, present in the same network. The *DeviceSpy* tool will be used in the local and remote machines to measure times of response.

In the image below, an illustration of how the system works regarding the case study is presented. The sensors from the *Robotic RTV Arm* are exposed to the network as *UPnP Devices*. The *SmartComponent UPnP Control Point* 5.2.4 scans the sensors and converts them into *OSGi UPnP Device* instances, next, the *SmartComponent*

Device Management 5.2.3 converts the previous instances in *SmartComponent Device Services*. For each complex service created, exists a correspondent representation as an *UPnP Device* in the network, as soon as an instantiation of a complex service is performed a representation in the network is reflected, the component responsible for export the devices is the *SmartComponent UPnP Exporter* 5.2.6. The idea of this image is to conceptually give an idea of how the information flow, since its measured, until is processed and turned into high level information to be consumed again for superior entities or actors.

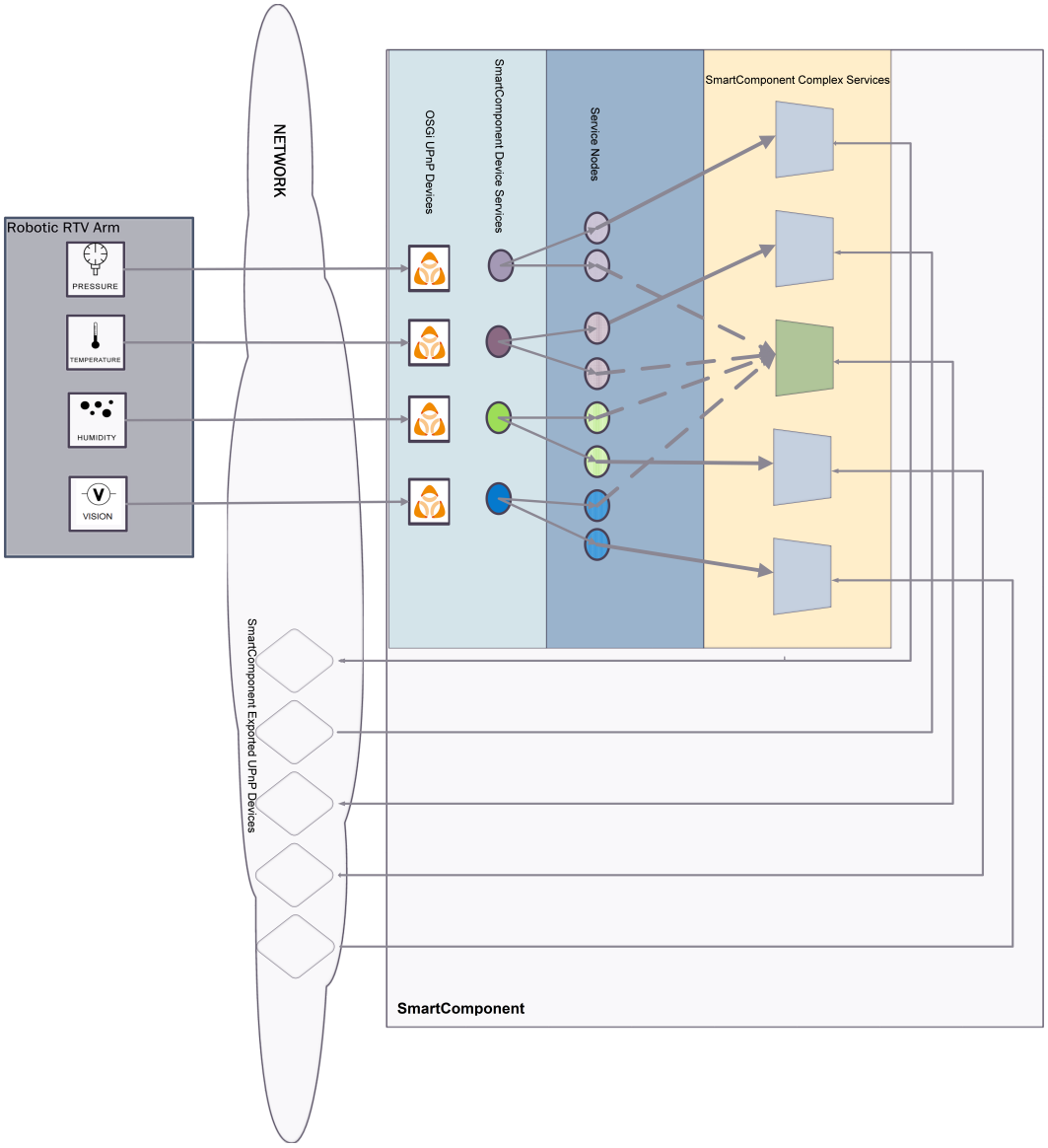


Figure 6.1: Case Study System Operation

6.3 Hypothesis Validation

Response time of invokation actions , aggregates the time that takes a *M2M* communication. In this case, is the time that takes to, process a request within the architecture, summed with the time that takes, messages of request and response to be delivered. To validate the hypothesis, this metric will be used with three previosuly explained functions of the architecture 5.2.8.1, ***GetLastResult()*** , ***GetSnapshot()*** and ***ListProviders()*** actions.

At scale of component, no problems where faced, validation and aggregation services sampling rates correct. Invocation of functions, as seen in the graphic below, was completed in milliseconds with success.

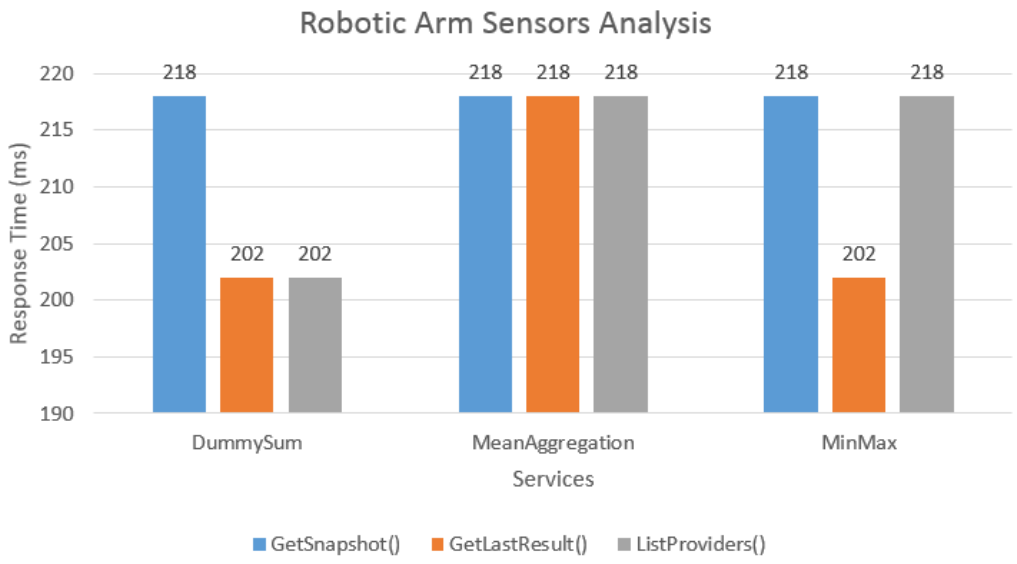
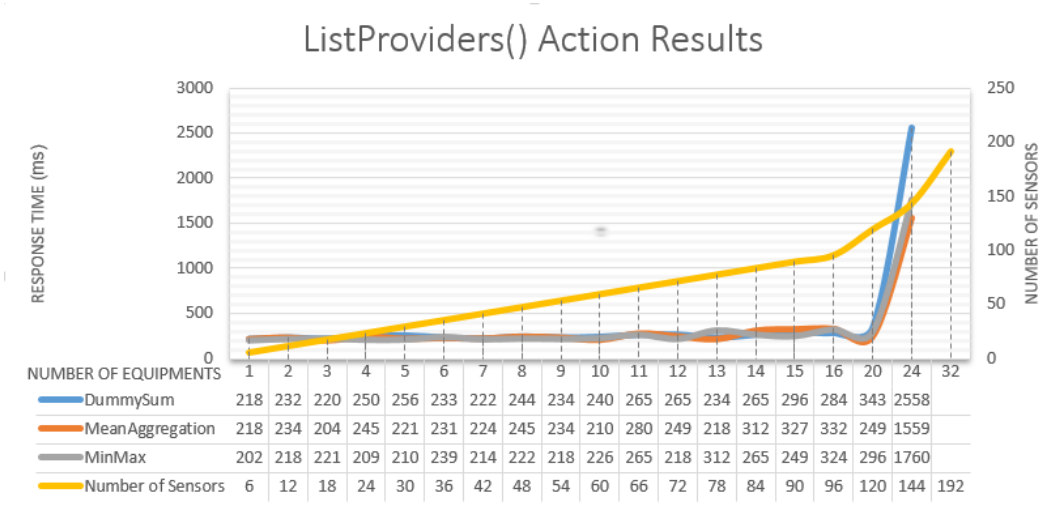


Figure 6.2: Robotic ARM Sensors Analysis Graph

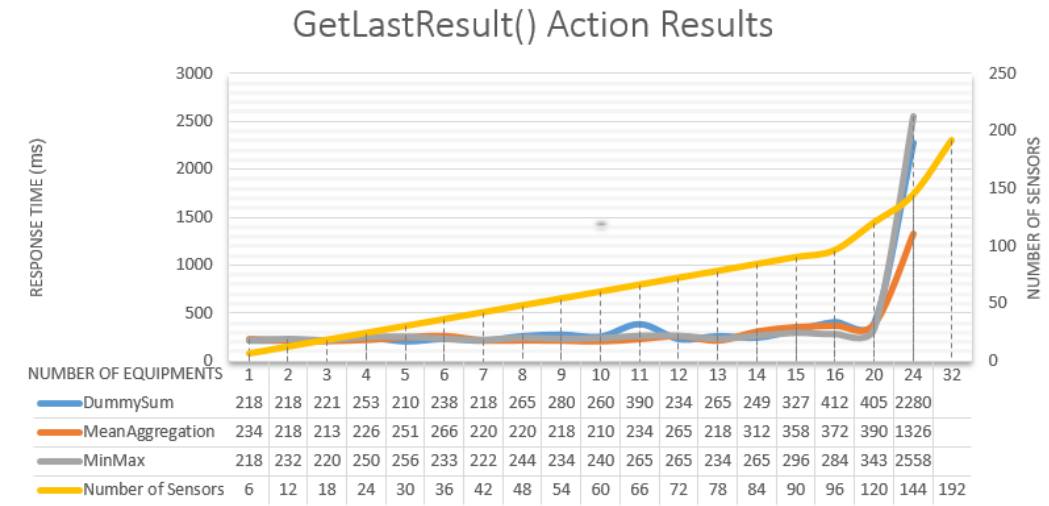
As the validation of the **Shop Floor** and **Supply Chain** cases is based in iterations of validation of a single **Equipment**, the tests were aggregated in the same graphs. The next three graphs show the times of response and number of services involved in the whole test process. Graphs are separated by functions for validation purposes. The ***GetSnapshot()*** action, internally to the architecture, involve a dequeue to every*ServiceNode* provider. Next, the data previously gathered, are processed and the output is sent to the remote invoker. Regarding the previous statement, in comparison with the other two functions, the internally processing effort is major, since the other two

just had to send data that is in memory. It was to be expected, higher response times to the ***GetSnapshot()*** function, compared to the other two functions, in case of internal to architecture malfunctions. As we can observe, all functions reply with similar response times. We can conclude, the delay is due to the network, the computation time within our application is not relevant to the response times.



NUMBER OF EQUIPMENTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
DummySum	218	232	220	250	256	233	222	244	234	240	265	265	234	265	296	284	343	2558	
MeanAggregation	218	234	204	245	221	231	224	245	234	210	280	249	218	312	327	332	249	1559	
MinMax	202	218	221	209	210	239	214	222	218	226	265	218	312	265	249	324	296	1760	
Number of Sensors	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	120	144	192

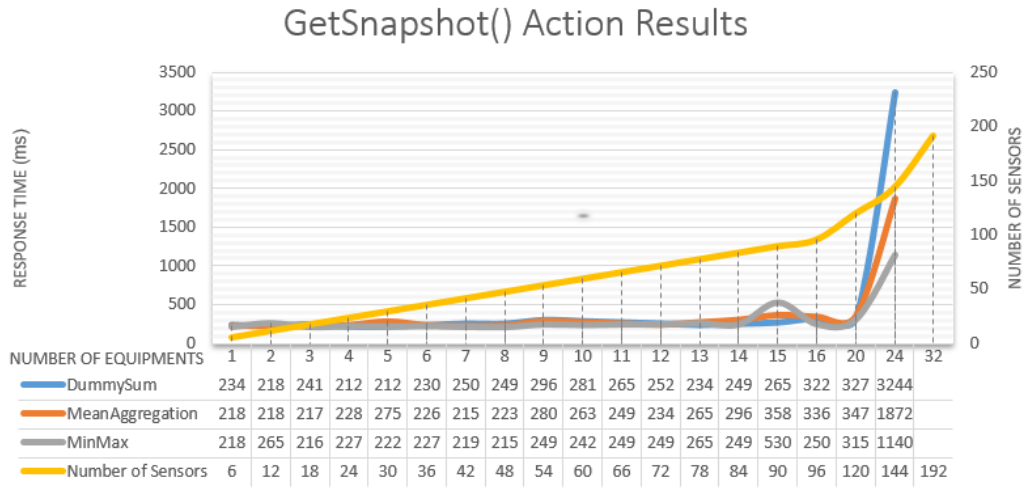
Figure 6.3: ListProviders() Action Results Graph



NUMBER OF EQUIPMENTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
DummySum	218	218	221	253	210	238	218	265	280	260	390	234	265	249	327	412	405	2280	
MeanAggregation	234	218	213	226	251	266	220	220	218	210	234	265	218	312	358	372	390	1326	
MinMax	218	232	220	250	256	233	222	244	234	240	265	265	234	265	296	284	343	2558	
Number of Sensors	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	120	144	192

Figure 6.4: GetLastResult() Action Results Graph

First fluctuations in the time that takes to detect devices, as formulated in hypothesis, has been detected when the number o sensors being exported by the remote machine



NUMBER OF EQUIPMENTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
DummySum	234	218	241	212	212	230	250	249	296	281	265	252	234	249	265	322	327	3244	
MeanAggregation	218	218	217	228	275	226	215	223	280	263	249	234	265	296	358	336	347	1872	
MinMax	218	265	216	227	222	227	219	215	249	242	249	249	265	249	530	250	315	1140	
Number of Sensors	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	120	144	192

Figure 6.5: GetSnapshot() Action Results Graph

has reached **54**. Corresponding to a number of **9** *Equipments* being monitored. Until this number was reached, no anomalies were detected, validating the hypothesis of monitoring a *Supply Chain* contitued of **8** *Equipments* in real time.

From **9** *Equipments*, to the maximum handled of **24**, the times of device detection vary between, **4 (min)** in the first case and **27 (min)** in the last case. However, as the device services appear in the remote machine, they were subscribed with success. The sampling rates, even in the last case, were maintained in **4 sec** for aggregation and **5 sec** for validation services. To enforce the previous statement, as we can observe in the graphs, response times of the actions were linear, with the architecture promptly replying to the requests. We conclude that the flooding of messages circulating in the network, were the main cause for the delay in devices detection. In the architecture, the control point, unlike in the *DeviceSpy* tool, the delay in sensor detection has become clear when the number of *Equipments* reached **20**.

Exchanging messages, has become totally unfeasible when the number of exported *Equipments* has reached **32**. Devices in both machines were detected and data subscriptions in both sides were done. In the remote machine, the values received by the complex services were *0*, meaning that queues were empty. The component *Smart-Component Logger* has reported ***java.net.SocketException: Connection reset*** throwed by the *Felix UPnP Driver* component. Meaning the *HTTP Socket* was not capable of

handle all the active connections. For a well-functioning proof of the architecture, the local *DeviceSpy* tool was able of discover and invoke the architecture services.

Analyzing the last two test iterations, we observe an exponential behavior. Relating that observation with the excessive consumption of resources, we can conclude that computation effort and the number of exported devices are directly related. The subset of protocols that composes the *UPnP* stack, requires regular message exchange, notification, update and event between *M2M*. Application *HTTP Sockets*, must handle a considerable amount of transactions and asynchronous messages, this leads to a substantial effort by the *CPU* to dispatch and acknowledge all the communications. Regarding to this, we can conclude that *UPnP* cannot scale in terms of communication to one single machine. The number of devices being exported - representing complex services in the architecture context – constitutes a problem. As we are not dealing with the communication directly, *Felix UPnP Driver* is the bridge between the architecture and the framework, this could represent additional overhead. The driver uses the *CyberDomo* library for *Java*, a possible solution could be to develop an architecture control point that uses the library directly. This way, we eliminate the necessity to recur to the framework as an intermediate tier and, possibly, achieve a better efficiency in communication issues.

Concluding, the reported results allow to validate, for real time needs, in all aspects (discovering and sampling frequency), the monitoring of **48** sensors. Exporting at that time, **20** complex services (**8** validation and **12** aggregation). In the case study perspective, a **Supply Chain** of **8** *Equipment's* could be monitored and controlled by our application. If the time that takes to detect devices is not a concern, this architecture was able to monitor and control **20** *Equipments*, with a required frequency sampling, only with slightly increased of the times in method invocation. Action response times has only passed the milliseconds threshold when handling **24** *Equipments*.

As the response times in 6.3 6.4 6.5 displayed a linear response time, until the number of **20** *Equipments* was reached, we suspected from hardware limitations and decided to develop a second test. The second test involved three machines, one containing the *SmartComponent* and the other two running the simulator. The machines running the simulator were exporting a total of **30** *Equipments*, one test iteration after the communication in the first test become unfeasible was done. The results observed

are displayed in the graph above.

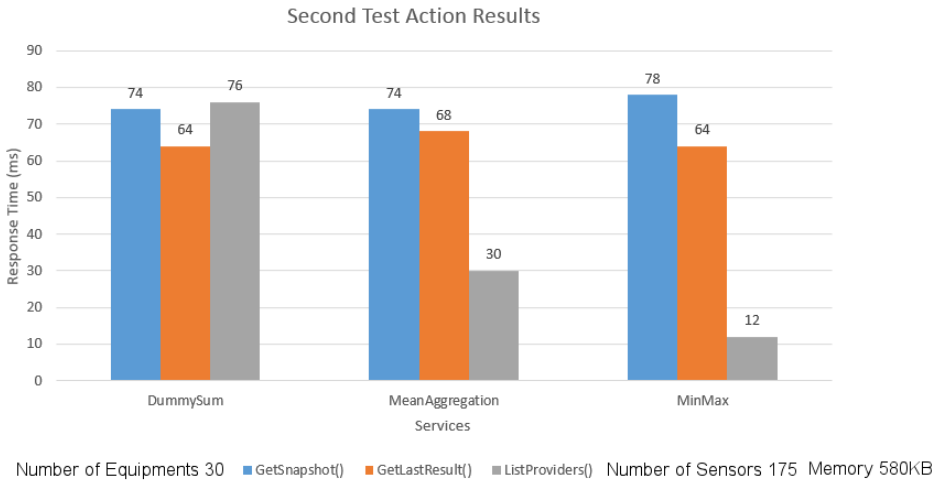


Figure 6.6: Second test action results.

The results observed, showed that the behavior is closely related with the hardware running the *SmartComponent*. The response times showed that, less than half of the average of the previous response times was achieved in this test. The exponential behavior that was previously observed, was due to the capacity of the hardware to handle large numbers of parallel connections, reinforcing a linear behavior from our architecture against the driver and hardware problems.

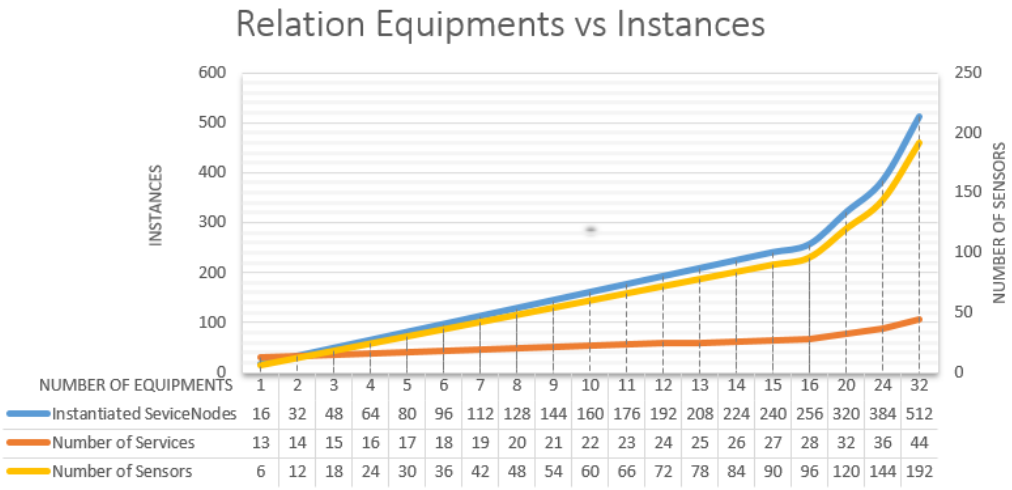
Regarding our architecture, the two tests have proven a correct function of the system. As we can see in the following section graphs, the memory consumption of the system is linear. Response to action invocation is linear and independent from the number of sensors being monitored 6.3 6.4 6.5. The number of complex services being exported and the number of sensors being subscribed, depends on the hardware as the network as well, as the cost of hardware is in constant decrease, we conclude the solution is feasible and covers all requirements of the application scenario.

In a concrete application scenario, regarding the first test, the system could be deployed, monitoring until **120** sensors in an efficiently way. If the hardware characteristic of the machine running the architecture are superior, the limit of **144** sensors can be exceeded, has showed in 6.6. Based in the first test hardware characteristics 6.4, a whole *Supply Chain* can be efficiently monitored, allowing for analysis, reconfiguration and deployment of complex service modules. Using our solution, in the case stated

in 6.2, **6** machines running the *SmartComponent* architecture, covers all the *Shop Floor* area. With all the area being analyzed, the *Smart Factory* concept is achieved, smart decisions over the production lifecycle are taken, culminating in an economic and resource enhanced efficiency.

6.4 Scalability

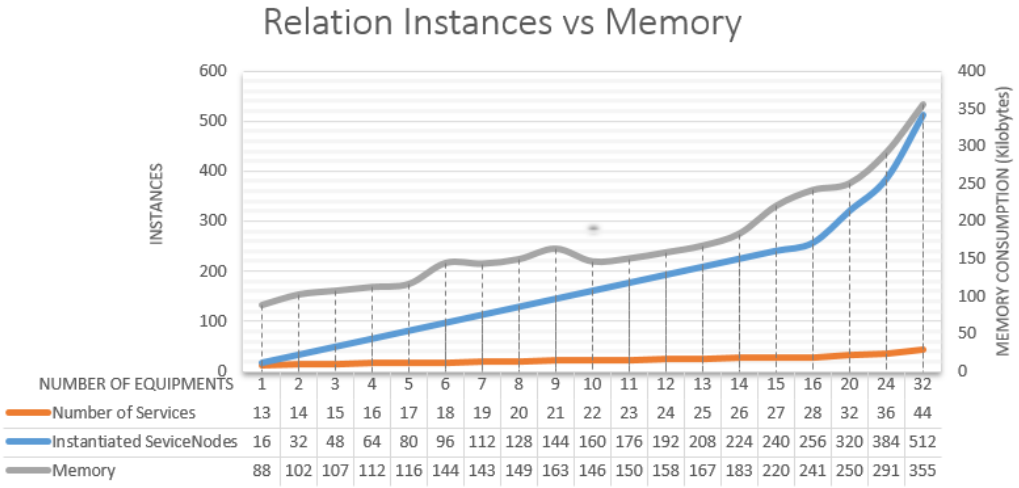
Action response times, have only passed the milliseconds threshold when handling **24** *Equipments* for the fist test. Introduce most capable hardware is a possibility for achieve scalability and adapt to the application scenario needs. Regarding to the previous possibility, we prove the architecture has resiliency to scalability issues. The next graphs, show how instances and the memory consumed by the *JVM* have evolved during the first test.



NUMBER OF EQUIPMENTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
Instantiated SeviceNodes	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	320	384	512
Number of Services	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	32	36	44
Number of Sensors	6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	120	144	192

Although the exportation of every complex services were no possible in the last test iteration of first test, we instantiated successfully in the local machine the required number of **44** complex services for analyses of **32** *Equipments*.

In the first test we achieved a number of **512** instantiated *Service Nodes* being used by **44** complex services. Taking in account other resources being consumed by the driver and the *Felix Framework* itself, we conclude that the evolution of memory consumption can be considered linear. The amount of resource consumption in the last test of ***355 Kilobytes*** is very acceptable regarding the effort in maintain the mentioned number of *ServiceNodes* and considering the number of complex service instances. In the



NUMBER OF EQUIPMENTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	20	24	32
Number of Services	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	32	36	44
Instantiated SeviceNodes	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	320	384	512
Memory	88	102	107	112	116	144	143	149	163	146	150	158	167	183	220	241	250	291	355

first test the local machine has ***4 GBytes*** of *RAM*, if the network were capable of handling more devices we could scale the application. The hardware characteristis of the two machines used in the test are: for the local machine, *Intel Core i5 2.27Ghz first generation CPU* with *4GBytes* of *RAM*; for the remote machine, *Intel Core i5 2.6Ghz third generation CPU* with *8GBytes* of *RAM* and the hosting operating systems were *Windows 7* in the local machine and *Windows 8* in the remote. In the two last test iterations, the simulation tool was consuming *98% - 100%* of *CPU* and *5GBytes* of *RAM*. Local machine, in the same iterations was consuming *60% of CPU* resources with irregular spontaneous pikes of *90% of CPU*. In the second test, the local machine hosting the *SmartComponent*, has the same specifications of the other two remote machines running the simulation tool. All machines have *Intel Core i7 2.4Ghz third generation CPUs* with *16GBytes* of *RAM* and the hosting operating system was *Windows 8*.

Chapter 7

Conclusion

The capability of obtain reliable and intelligent information, derived from the sensors data, was a major requirement of this work. Attached to previous requirement, with considerable importance, the capability of produce new models of analysis with a relative easiness , deploy them into the system and remove active ones on demand. In addiction, an active complex service has connection with external actors, the service is a network representation of an model instance. The capability of switch the correspondent instance in *runtime*, without breaking the connection to the external actor, eliminates the need of reconfigure the system whenever a newer model or version of a model is introduced in the system and exists the necessity of use that model. To make the system capable of interact with other systems and components, it must be capable of implement an adopted ontology, this way, it can manipulate and be manipulated by external entities. To expose the system functionalities, a protocol that is capable of announce the system capabilities, as turn the communication transparent to external agents, must be implemented. To handle the interactions previously refered, the system must deal with a considerable level of abstraction. In a technical perspective, this means the system must be capable of virtualize its services and treat external components as virtualizations as well.

Virtualization to a network of complex services exported by the application was successfully achieved recurring to *UPnP*. Previously pointed as a major concern 3.1, extract meaning from communication, was overcome through the implementation of *DIL* language.Due to the previous achievements, all services managed by the architecture,

become transparent and horizontal to external agents. Those agents can explore the functionalities of the complex services in a uniform way, using the *UPnPService* type and the inner functionalities to each service, declared in the respective *NSD* documents. The developed functionalities of the architecture and its complex services, allow inter device communication, consequently, fostering synergies with other actors, eg, personnel at the *Shop Floor* using portable monitoring stations. The functionalities exposed by the *UPnP* services to the network 5.10 5.15, allows for a flexible reconfiguration of the complex services and *VSIG* formation; that way, was overcome the consideration pointed at 3.4, allowing for an smart logistic of the services logic. Data acquisition, event or frequency based 2.6, was successfully accomplished, due to the possibility of subscribe *UPnP State Variables*. Collaboration between *Smart-Component* nodes is guaranteed. The encapsulation of the complex services as well defined *NetDev* services 5.2.4, allows for both nodes identify the complex services and perform cross check validation of data. This previous characteristic, allow for a cloud infrastructure to subscribe and manage the services being exported, well filling an important need stated the project 2.4. A configuration service, was integrated in the architecture, validating the modularity needs 3.5. This previous characteristic, allow for a sensor cloud infrastructure to subscribe and manage the services being exported by the archtecture, this way, filling an important need of the *SelSus* project 2.4. A configuration service, was integrated in the architecture, validating the modularity needs 3.5. *On the fly* integration of data analyses modules was achieved, reinforcing the modularity nature of the architecture, allowing mitigate the cost time effective needs of programing and deploying new software components over the equipment's in the production line. Finally, the architecture was validated against a real case based scenario of application. Summing up, we conclude that a significant contribution in terms of literature and application of technologies was done, resulting in a step forward regarding *Smart Factories* to become a reality.

7.1 Future work

Based on the results achieved and on the insights gained during this work, the following list of possible lines of work have been identified:

- Increase of the available set of actions that can be performed over a complex service.

  Rather than just pull output from services, implement capabilities like, eg. introduce real time data from another device to be processed.

- Study and implementation of new types of analysis modules, increasing the number of complex analysis services that can be performed by the system.

  Use Neuronal or Bayesian networks to enrich the prediction capabilities of the system.

- Develop a proprietary *UPnP* driver component, it will create a direct control over the devices being exported and imported.

  The result will be an increase in the number of connections a single machine can handle.

- Create and export an architecture *Logger Component* as *UPnP* device.

  Subscribing that device, will allow for treatment of notifications related with the architecture (eg. Alert of a sensor service requested for consuming, that does not exists).

- Increase the registry search specifications.

  When assigning a provider to a consumer or instantiating a service (eg. Select a temperature service with a specific accuracy of measurement).

With pervasive computing increase, the *IoT* has about to become a reality. *Fiware* [29], is a platform that aims to cement its position in the available range of solutions that wants to bring *IoT* close to end customers and enterprises. This open set of specifications, ends with a structure that allows for a strongly coupled interaction between service providers and service consumers, all benefiting from the interactions. In the Industrial context, this platform allows for a factory to expose its services (eg. monitoring

systems and sensors) to the developers that want to produce and test new solutions. An *API* grants the cost-effective development of new services. The virtualized company infrastructures, are available to be used as concrete test cases to the developers trough a cloud [30] of *"well-defined Service End Points"*.

Integrate this platform *API* is part of our future work, we will benefit from this integration as we can progress in the developments, supported by concrete cases of application. Moreover, *Fiware* is part of the *Seventh Framework Programme* as well, together with the projects partners, we will help to foment the competitiveness in the European Industry.

Appendices

Appendix A

Abbreviations

M2M Machine to Machine

IoT Internet of Things

UPnP Universal Plug & Play

WSN Wireless Sensor Networks

IWSN Industrial Wireless Sensor Networks

SOA Service Oriented Architecture

SOC Service Oriented Computing

OO Object Oriented

POJO Plain Old Java Object

RF Radio Frequency

OSI Open Systems Interconnection Model

XML Xtensible Markup Language

WSDL Web Service Description Language

UDDI Universal Description Discovery and Integration

VSIG Virtual Sensor Information Group

JAR Java ARchive

NETDEV NETwork-enabled DEVice

NSD NetDev Self Description

TDD Task Description Document

QRD Quality Result Document

TFD Task Fulfilment Document

Appendix B

Components NSD

```
1 <?xml version="1.0"?>
2
3   <description>
4     <name>ServiceController</name>
5     <model>1.0</model>
6     <manufacturer>Luis</manufacturer>
7     <serial>3001-50-22-3000</serial>
8     <built>2014-01-30T09:00:00</built>
9     <physicalProperties />
10  </description>
11
12  <!-- The task range contains a list of task definitions, which can be
        executed
13    by this netdev -->
14  <taskRange>
15    <taskDefinition name="DisposeService" sntd="/task/netdev/logical/
          service/controller/dispose_service">
16      <dataDefinitions>
17        <!-- Input -->
18        <dataDefinition id="ServiceUID">
19          <metaData name="datatype">integer</metaData>
20          <metaData name="minValue">1</metaData>
21          <metaData name="maxValue">99999999999</metaData>
22          <metaData name="unit">integer</metaData>
23        </dataDefinition>
24        <!-- Output -->
```

```
25          <dataDefinition id="ActionResult">
26            <metaData name="datatype">string</metaData>
27            <metaData name="unit">string</metaData>
28          </dataDefinition>
29        </dataDefinitions>
30        <tddSchema>
31          <goals>
32            <goal dataDefinitionRef="ServiceUID" />
33          </goals>
34        </tddSchema>
35        <qrdSchema>
36          <result dataDefinitionRef="ActionResult" />
37        </qrdSchema>
38      </taskDefinition>
39          <taskDefinition name="ListServices" sntd="/task/netdev/logical/
                service/controller/list_services">
40        <dataDefinitions>
41          <!-- Output -->
42          <dataDefinition id="SensorServices">
43            <metaData name="datatype">string</metaData>
44            <metaData name="unit">json</metaData>
45          </dataDefinition>
46          <dataDefinition id="ComplexServices">
47            <metaData name="datatype">string</metaData>
48            <metaData name="unit">json</metaData>
49          </dataDefinition>
50        </dataDefinitions>
51        <tddSchema>
52          <goals/>
53        </tddSchema>
54        <qrdSchema>
55          <result dataDefinitionRef="SensorServices" />
56          <result dataDefinitionRef="ComplexServices" />
57        </qrdSchema>
58      </taskDefinition>
59    </taskRange>
60    <localizations/>
61 </nsd>
```

ServiceController NSD Document

```
<?xml version="1.0"?>

  <description>
    <name>ServiceFactory</name>
    <model>1.0</model>
    <manufacturer>Luis</manufacturer>
    <serial>3001-50-22-3000</serial>
    <built>2014-01-30T09:00:00</built>
    <physicalProperties />
  </description>

  <taskRange>
    <taskDefinition name="InstantiateServiceFrequencyData" sntd="/task/
        netdev/logical/service/factory/InstantiateServiceFrequencyData">
      <dataDefinitions>
        <!-- Input -->
        <dataDefinition id="ServiceID">
          <metaData name="datatype">string</metaData>
          <metaData name="unit">string</metaData>
        </dataDefinition>
        <dataDefinition id="ServiceFrequency">
          <metaData name="datatype">integer</metaData>
          <metaData name="minValue">500</metaData>
          <metaData name="maxValue">100000</metaData>
          <metaData name="unit">millisenconds</metaData>
        </dataDefinition>
        <dataDefinition id="SensorType">
          <metaData name="datatype">string</metaData>
          <metaData name="unit">array</metaData>
        </dataDefinition>
        <dataDefinition id="SensorModel">
          <metaData name="datatype">string</metaData>
          <metaData name="unit">array</metaData>
        </dataDefinition>
        <dataDefinition id="SensorUID">
          <metaData name="datatype">integer</metaData>
          <metaData name="unit">array</metaData>
        </dataDefinition>
        <dataDefinition id="ComplexType">
```

```
39          <metaData name="datatype">string</metaData>
40          <metaData name="unit">array</metaData>
41        </dataDefinition>
42        <dataDefinition id="ComplexUID">
43          <metaData name="datatype">string</metaData>
44          <metaData name="unit">array</metaData>
45        </dataDefinition>
46
47        <!-- Output -->
48        <dataDefinition id="ActionResult">
49          <metaData name="datatype">string</metaData>
50          <metaData name="unit">string</metaData>
51        </dataDefinition>
52      </dataDefinitions>
53
54      <tddSchema>
55        <goals>
56          <goal dataDefinitionRef="ServiceID" />
57          <goal dataDefinitionRef="ServiceFrequency" />
58        </goals>
59        <boundaryConditions>
60          <boundaryCondition dataDefinitionRef="SensorType" />
61          <boundaryCondition dataDefinitionRef="SensorModel" />
62          <boundaryCondition dataDefinitionRef="SensorUID" />
63          <boundaryCondition dataDefinitionRef="ComplexType" />
64          <boundaryCondition dataDefinitionRef="ComplexUID" />
65        </boundaryConditions>
66      </tddSchema>
67      <qrdSchema>
68        <result dataDefinitionRef="ActionResult" />
69      </qrdSchema>
70    </taskDefinition>
71    <taskDefinition name="InstantiateServiceAccumulationCycles" sntd="/
          task/netdev/logical/service/factory/
          InstantiateServiceAccumulationCycles">
72      <dataDefinitions>
73        <!-- Input -->
74        <dataDefinition id="ServiceID">
75          <metaData name="datatype">string</metaData>
76          <metaData name="unit">string</metaData>
```

```
77          </dataDefinition>
78          <dataDefinition id="AccumulationCycles">
79            <metaData name="datatype">integer</metaData>
80            <metaData name="minValue">1</metaData>
81            <metaData name="maxValue">100000</metaData>
82            <metaData name="unit">millisenconds</metaData>
83          </dataDefinition>
84          <dataDefinition id="SensorType">
85            <metaData name="datatype">string</metaData>
86            <metaData name="unit">array</metaData>
87          </dataDefinition>
88          <dataDefinition id="SensorModel">
89            <metaData name="datatype">string</metaData>
90            <metaData name="unit">array</metaData>
91          </dataDefinition>
92          <dataDefinition id="SensorUID">
93            <metaData name="datatype">integer</metaData>
94            <metaData name="unit">array</metaData>
95          </dataDefinition>
96          <dataDefinition id="ComplexType">
97            <metaData name="datatype">string</metaData>
98            <metaData name="unit">array</metaData>
99          </dataDefinition>
100         <dataDefinition id="ComplexUID">
101           <metaData name="datatype">string</metaData>
102           <metaData name="unit">array</metaData>
103         </dataDefinition>
104         <!-- Output -->
105         <dataDefinition id="SensorServices">
106           <metaData name="datatype">string</metaData>
107           <metaData name="unit">json</metaData>
108         </dataDefinition>
109         <dataDefinition id="ActionResult">
110           <metaData name="datatype">string</metaData>
111           <metaData name="unit">string</metaData>
112         </dataDefinition>
113       </dataDefinitions>
114
115       <tddSchema>
116         <goals>
```

```
117           <goal dataDefinitionRef="ServiceID" />
118           <goal dataDefinitionRef="AccumulationCycles" />
119         </goals>
120         <boundaryConditions>
121           <boundaryCondition dataDefinitionRef="SensorType" />
122           <boundaryCondition dataDefinitionRef="SensorModel" />
123           <boundaryCondition dataDefinitionRef="SensorUID" />
124           <boundaryCondition dataDefinitionRef="ComplexType" />
125           <boundaryCondition dataDefinitionRef="ComplexUID" />
126         </boundaryConditions>
127       </tddSchema>
128
129       <qrdSchema>
130         <result dataDefinitionRef="ActionResult" />
131       </qrdSchema>
132     <taskDefinition name="ListAvailableServices" sntd="/task/netdev/
          logical/service/factory/ListAvailableServices">
133       <dataDefinitions>
134
135         <!-- Output -->
136         <dataDefinition id="SensorServices">
137           <metaData name="datatype">string</metaData>
138           <metaData name="unit">json</metaData>
139         </dataDefinition>
140         <dataDefinition id="ActionResult">
141           <metaData name="datatype">string</metaData>
142           <metaData name="unit">string</metaData>
143         </dataDefinition>
144       </dataDefinitions>
145
146       <tddSchema>
147         <goals>
148           <goal dataDefinitionRef="ServiceID" />
149           <goal dataDefinitionRef="AccumulationCycles" />
150         </goals>
151       </tddSchema>
152
153       <qrdSchema>
154         <result dataDefinitionRef="ActionResult" />
155       </qrdSchema>
```

```
156     </taskDefinition>
157   </taskRange>
158   <localizations/>
159
160 </nsd>
```

ServiceFactory NSD Document

```
1 <?xml version="1.0"?>
2
3   <description>
4     <name>SmartComponentService</name>
5     <model>1.0</model>
6     <manufacturer>Luis</manufacturer>
7     <serial>3001-50-22-3000</serial>
8     <built>2014-01-30T09:00:00</built>
9     <physicalProperties />
10  </description>
11
12  <taskRange>
13    <taskDefinition name="AddConsumer" sntd="/task/netdev/logical/service
          /smartcomponent_service/add_consumer">
14        <!-- Input -->
15        <dataDefinition id="ComplexUID">
16          <metaData name="datatype">integer</metaData>
17          <metaData name="minValue">1</metaData>
18          <metaData name="maxValue">9999999999999</metaData>
19          <metaData name="unit">integer</metaData>
20        </dataDefinition>
21        <!-- Output -->
22        <dataDefinition id="ActionResult">
23          <metaData name="datatype">string</metaData>
24          <metaData name="unit">string</metaData>
25        </dataDefinition>
26      </dataDefinitions>
27
28      <tddSchema>
29        <goals>
30          <goal dataDefinitionRef="ComplexUID" />
31        </goals>
32        <boundaryConditions/>
```

```
33      </tddSchema>
34
35      <qrdSchema>
36        <result dataDefinitionRef="ActionResult" />
37      </qrdSchema>
38    </taskDefinition>
39
40    <taskDefinition name="AddProvider" sntd="/task/netdev/logical/service
          /smartcomponent_service/add_provider">
41      <!-- Input -->
42      <dataDefinition id="ComplexUID">
43        <metaData name="datatype">integer</metaData>
44        <metaData name="minValue">1</metaData>
45        <metaData name="maxValue">9999999999999</metaData>
46        <metaData name="unit">integer</metaData>
47      </dataDefinition>
48      <dataDefinition id="SensorUID">
49        <metaData name="datatype">integer</metaData>
50        <metaData name="minValue">1</metaData>
51        <metaData name="maxValue">9999999999999</metaData>
52        <metaData name="unit">integer</metaData>
53      </dataDefinition>
54      <!-- Output -->
55      <dataDefinition id="ActionResult">
56        <metaData name="datatype">string</metaData>
57        <metaData name="unit">string</metaData>
58      </dataDefinition>
59    </dataDefinitions>
60
61
62    <tddSchema>
63      <goals>
64        <goal dataDefinitionRef="ComplexUID" />
65        <goal dataDefinitionRef="SensorUID" />
66      </goals>
67      <boundaryConditions/>
68    </tddSchema>
69
70    <qrdSchema>
71      <result dataDefinitionRef="ActionResult" />
```

```
72      </qrdSchema>
73
74    <taskDefinition name="GetLastResult" sntd="/task/netdev/logical/
          service/smartcomponent_service/get_last_result>
75
76        <!-- Output -->
77        <dataDefinition id="ActionResult">
78          <metaData name="datatype">string </metaData>
79          <metaData name="unit">string </metaData>
80        </dataDefinition>
81      </dataDefinitions>
82
83      <tddSchema>
84        <goals/>
85        <boundaryConditions/>
86      </tddSchema>
87
88      <qrdSchema>
89        <result dataDefinitionRef="ActionResult" />
90      </qrdSchema>
91
92    </taskDefinition>
93    <taskDefinition name="GetSnapshot" sntd="/task/netdev/logical/service
          /smartcomponent_service/get_snapshot>
94        <!-- Output -->
95        <dataDefinition id="ActionResult">
96          <metaData name="datatype">string</metaData>
97          <metaData name="unit">string</metaData>
98        </dataDefinition>
99      </dataDefinitions>
100
101     <tddSchema>
102       <goals/>
103       <boundaryConditions/>
104     </tddSchema>
105
106     <qrdSchema>
107       <result dataDefinitionRef="ActionResult" />
108     </qrdSchema>
109
```

```
110     </taskDefinition>
111     <taskDefinition name="ListProviders" sntd="/task/netdev/logical/
            service/smartcomponent_service/list_providers>
112         <!-- Output -->
113         <dataDefinition id="ActionResult">
114           <metaData name="datatype">string</metaData>
115           <metaData name="unit">string</metaData>
116         </dataDefinition>
117       </dataDefinitions>
118
119       <tddSchema>
120         <goals/>
121         <boundaryConditions/>
122       </tddSchema>
123
124       <qrdSchema>
125         <result dataDefinitionRef="ActionResult" />
126       </qrdSchema>
127
128     </taskDefinition>
129     <taskDefinition name="RemoveProvider" sntd="/task/netdev/logical/
            service/smartcomponent_service/remove_provider>
130         <!-- Input -->
131         <dataDefinition id="ComplexUID">
132           <metaData name="datatype">integer</metaData>
133           <metaData name="minValue">1</metaData>
134           <metaData name="maxValue">9999999999999</metaData>
135           <metaData name="unit">integer</metaData>
136         </dataDefinition>
137         <dataDefinition id="SensorUID">
138           <metaData name="datatype">integer</metaData>
139           <metaData name="minValue">1</metaData>
140           <metaData name="maxValue">9999999999999</metaData>
141           <metaData name="unit">integer</metaData>
142         </dataDefinition>
143         <!-- Output -->
144         <dataDefinition id="ActionResult">
145           <metaData name="datatype">string</metaData>
146           <metaData name="unit">string</metaData>
147         </dataDefinition>
```

```
148         </dataDefinitions>
149
150         <tddSchema>
151           <goals>
152             <goal dataDefinitionRef="ComplexUID" />
153             <goal dataDefinitionRef="SensorUID" />
154           </goals>
155         </tddSchema>
156
157         <qrdSchema>
158           <result dataDefinitionRef="ActionResult" />
159         </qrdSchema>
160
161       </taskDefinition>
162     </taskRange>
163     <localizations/>
164 </nsd>
```

ComplexService NSD Document

References

[1] I-RAMP3, “I- ramp3 - inteligent reconfigurable machines for smart plug&produce production,” http://www.i-ramp3.eu/, 2014 (accessed September 24, 2014).

[2] SelSus, “Selsus - health monitoring and life-long capability management for self- sustaining manufacturing systems,” ""http://www.selsus.eu/", 2014 (accessed September 24, 2014).

[3] Z. M. Bi, S. Y. Lang, W. Shen, and L. Wang, “Reconfigurable manufacturing systems: the state of the art,” International Journal of Production Research, vol. 46, no. 4, pp. 967–992, 2008.

[4] UPnP, “Upnp forum,” ""http://www.upnp.org/", 2014 (accessed September 24, 2014).

[5] F. Grow, “Plug things framework,” ""http://www.freedomgrow.pt/fg/en-gb/solutions/developmentframework.aspx", 2014 (accessed September 24, 2014).

[6] V. C. Gungor and G. P. Hancke, “Industrial wireless sensor networks: Challenges, design principles, and technical approaches,” Industrial Electronics, IEEE Transactions on, vol. 56, no. 10, pp. 4258–4265, 2009.

[7] J. Akerberg, M. Gidlund, and M. Bjorkman, “Future research challenges in wireless sensor and actuator networks targeting industrial automation,” in Industrial Informatics (INDIN), 2011 9th IEEE International Conference on. IEEE, 2011, pp. 410–415.

[8] WirelessHART, “Wirelesshart overview,” ""http://en.hartcomm.org/hcp/tech/wihart/wireless_overview.html", 2014 (accessed September 24, 2014).

[9] J. M. Prinsloo, C. L. Schulz, D. G. Kourie, W. Theunissen, T. Strauss, R. Van Den Heever, and S. Grobbelaar, “A service oriented architecture for wireless sensor and actor network applications,” in Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. South African Institute for Computer Scientists and Information Technologists, 2006, pp. 145–154.

[10] J. Leguay, M. Lopez-Ramos, K. Jean-Marie, and V. Conan, “An efficient service oriented architecture for heterogeneous and dynamic wireless sensor networks,” in Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on. IEEE, 2008, pp. 740–747.

[11] J. Architecture, “Jini architecture specification,” ""https://river.apache.org/doc/specs/html/jini-spec.html", 2014 (accessed September 24, 2014).

[12] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, “Looci: a loosely-coupled component infrastructure for networked embedded systems,” in Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia. ACM, 2009, pp. 195–203.

[13] D. Hughes, M. Daudé, G. Coulson, G. Blair, P. Smith, K. Beven, and W. Tych, “Managing heterogeneous data flows in wireless sensor networks using a" split personality’mote platform,” in Applications and the Internet, 2008. SAINT 2008. International Symposium on. IEEE, 2008, pp. 145–148.

[14] E. Avilés-López and J. A. García-Macías, “Tinysoa: a service-oriented architecture for wireless sensor networks,” Service Oriented Computing and Applications, vol. 3, no. 2, pp. 99–108, 2009.

[15] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg, “Implementation of coap and its application in transport logistics,” Proc. IP+ SN, Chicago, IL, USA, 2011.

[16] CoRE, “Constrained restful environments,” ""http://tools.ietf.org/wg/core/", 2014 (accessed September 24, 2014).

[17] P. Evensen and H. Meling, “Sensor virtualization with self-configuration and

flexible interactions," in Proceedings of the 3rd ACM International Workshop on Context-Awareness for Self-Managing Systems. ACM, 2009, pp. 31–38.

[18] A. Bottaro, A. Gérodolle, and P. Lalanda, "Pervasive service composition in the home network," in Advanced Information Networking and Applications, 2007. AINA'07. 21st International Conference on. IEEE, 2007, pp. 596–603.

[19] L. Schor, P. Sommer, and R. Wattenhofer, "Towards a zero-configuration wireless sensor network architecture for smart buildings," in Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings. ACM, 2009, pp. 31–36.

[20] A. Sleman and R. Moeller, "Integration of wireless sensor network services into other home and industrial networks; using device profile for web services (dpws)," in Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on. IEEE, 2008, pp. 1–5.

[21] V. Sousa, "Cobertura, conectividade e requisitos de uma rede de sensores sem fios," 2014.

[22] SOCRADES, "Socrades union's 6th framework programme." ""http://www.socrades.eu/Home/default.html", 2014 (accessed September 24, 2014).

[23] OSGi, "Osgi aliance," ""http://www.osgi.org/Main/HomePage", 2014 (accessed September 24, 2014).

[24] T. A. S. Foundation, "Apache felix framework," ""https://felix.apache.org/", 2014 (accessed September 24, 2014).

[25] W. J. Gédéon, OSGi and Apache Felix 3.0 Beginner's Guide. Packt Publishing Ltd, 2010.

[26] C. Escoffier, R. S. Hall, and P. Lalanda, "ipojo: An extensible service-oriented component framework," in Services Computing, 2007. SCC 2007. IEEE International Conference on. IEEE, 2007, pp. 474–481.

[27] T. A. S. Foundation, "Apache felix upnp basedriver," ""https://felix.apache.org/site/apache-felix-upnp.html", 2014 (accessed September 24, 2014).

[28] K. Thoelen, N. Matthys, W. Horré, C. Huygens, W. Joosen, D. Hughes, L. Fang,

and S.-U. Guan, “Supporting reconfiguration and re-use through self-describing component interfaces,” in Proceedings of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks. ACM, 2010, pp. 29–34.

[29] Fiware, “Fiware (core platform of the future internet),” ""http://www.fi-ware.org/about/", 2014 (accessed September 24, 2014).

[30] FiLab, “Fiware lab,” ""https://account.lab.fi-ware.org/", 2014 (accessed September 24, 2014).